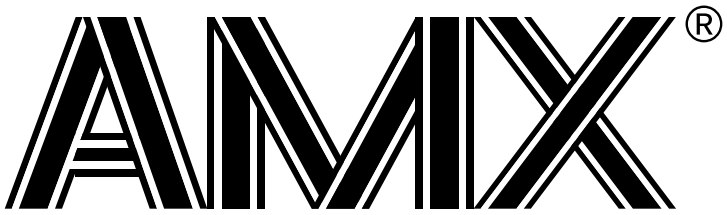


AMX™ 86 User's Guide

First Printing: **November 1, 1990**
Last Printing: **March 1, 2005**

Copyright © 1990 - 2005

KADAK Products Ltd.
206 - 1847 West Broadway Avenue
Vancouver, BC, Canada, V6J 1Y5
Phone: (604) 734-2796
Fax: (604) 734-8114

TECHNICAL SUPPORT

KADAK Products Ltd. is committed to technical support for its software products. Our programs are designed to be easily incorporated in your systems and every effort has been made to eliminate errors.

Engineering Change Notices (ECNs) are provided periodically to repair faults or to improve performance. You will automatically receive these updates during the product's initial support period. For technical support beyond the initial period, you must purchase a Technical Support Subscription. Contact KADAK for details. Please keep us informed of the primary user in your company to whom update notices and other pertinent information should be directed.

Should you require direct technical assistance in your use of this KADAK software product, engineering support is available by telephone, fax or e-mail. KADAK reserves the right to charge for technical support services which it deems to be beyond the normal scope of technical support.

We would be pleased to receive your comments and suggestions concerning this product and its documentation. Your feedback helps in the continuing product evolution.

KADAK Products Ltd.
206 - 1847 West Broadway Avenue
Vancouver, BC, Canada, V6J 1Y5

Phone: (604) 734-2796
Fax: (604) 734-8114
e-mail: amxtech@kadak.com

Copyright © 1990-2005 by KADAK Products Ltd.
All rights reserved.

No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language or computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual or otherwise, without the prior written permission of KADAK Products Ltd., Vancouver, B.C., CANADA.

DISCLAIMER

KADAK Products Ltd. makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability and fitness for any particular purpose. Further, KADAK Products Ltd. reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of KADAK Products Ltd. to notify any person of such revision or changes.

TRADEMARKS

AMX in the stylized form and KwikNet are registered trademarks of KADAK Products Ltd.
AMX, AMX/FS, InSight, *KwikLook* and KwikPeg are trademarks of KADAK Products Ltd.
Microsoft, MS-DOS and Windows are registered trademarks of Microsoft Corporation.
All other trademarked names are the property of their respective owners.

AMX 86 USER'S GUIDE

Table of Contents

Page

Section 1: System Description

1. AMX Overview **1**

1.1 Introduction 1
1.2 Glossary 3
1.3 AMX Nomenclature 7

2. General AMX Operation **9**

2.1 Introduction to Multitasking 9
2.2 AMX Operation 11
2.3 AMX Managers 17
2.4 Starting AMX 20

3. Application Tasks **25**

3.1 Task Creation 25
3.2 Task States 27
3.3 Starting a Task 29
3.4 Task Priority 30
3.5 Task Execution 31
3.6 Task and Event Synchronization 32
3.7 Task Timing 34
3.8 Ending a Task 35
3.9 Message Passing 36
3.10 Restart Procedures 42
3.11 Exit Procedures 44
3.12 Task Enhancements 46

4. Interrupt Service Procedures **49**

4.1 The Processor Interrupt Facility 49
4.2 ISPs for External Interrupts 51
4.3 Nested Interrupts 56
4.4 ISP/Task Communication 57
4.5 Task Error Traps 60
4.6 Non-Maskable Interrupt 64
4.7 Special Interrupts 65
4.8 Vector Table Initialization 68

5. AMX Timing Control **69**

5.1 Introduction to Timing Facilities 69
5.2 AMX Clock Handler and Kernel Task 71
5.3 Interval Timers and Timer Procedures 75
5.4 Task Time Slicing 79
5.5 Time/Date Manager 82

Page

Section 1: System Description (Cont'd.)

6. AMX Semaphore Manager **91**

6.1 Introduction 91
6.2 Semaphore Use 93
6.3 Semaphore Applications 97

7. AMX Event Manager **103**

7.1 Introduction 103
7.2 Event Synchronization 105
7.3 Event Flag Application 107

8. AMX Message Exchange Manager **111**

8.1 Introduction 111
8.2 Message Exchange Use 113
8.3 Message Exchange Application 115

9. AMX Buffer Manager **119**

9.1 Introduction 119
9.2 Buffer Pool Use 120
9.3 Buffer Applications 122
9.4 Buffer Manager Caveats 124

10. AMX Memory Manager **125**

10.1 Introduction 125
10.2 Nomenclature 127
10.3 Memory Allocation 128
10.4 Private Memory Allocation 130
10.5 Memory Assignment 131
10.6 Memory Assignment Procedure 132

11. AMX Circular List Manager **135**

11.1 Circular Lists 135
11.2 Circular List Use 136
11.3 Circular List Structure 137

AMX 86 USER'S GUIDE
Table of Contents (Cont'd.)

Page

Section 1: System Description (Cont'd.)

12. AMX Linked List Manager **139**

12.1 Introduction 139
12.2 Linked Lists 140
12.3 Linked List Use 142

13. Advanced Topics **147**

13.1 Fatal Exit 147
13.2 User Error Procedure 150
13.3 Task Scheduling Hooks 152
13.4 Abnormal Task Termination 153
13.5 Task Suspend/Resume 158
13.6 Breakpoint Manager 159

Section 2: System Construction

14. AMX System Configuration **163**

14.1 System Configuration Module 163
14.2 System Configuration Builder 164
14.3 Using the Builder 166
14.4 System Parameter Definition 170
14.5 AMX Object Allocation 173
14.6 Restart/Exit Procedure Definition 176
14.7 Task Definition 178
14.8 Timer Definition 181
14.9 Semaphore Definition 183
14.10 Event Group Definition 185
14.11 Message Exchange Definition 187
14.12 Buffer Pool Definition 189
14.13 Breakpoint Manager Definition 191

Section 3: Programming Guide

15. AMX Service Procedures **193**

15.1 Introduction 193
15.2 Summary of Services 194

16. AMX 86 Procedures **201**

16.1 Introduction 201
Alphabetic List of Procedures 205

AMX 86 USER'S GUIDE
Table of Contents (Cont'd.)

Page

Appendices

A. AMX 86 Reserved Words **347**

B. AMX 86 Error Codes **349**

C. Configuration Generator Specifications **353**

C.1 Introduction 353
C.2 User Parameter File Specification 354
C.3 System Configuration Template 359
C.4 Porting the Configuration Generator 362

D. AMX 86 Structure and Constant Definitions **365**

D.1 AMX C Structures and Constants 365
D.2 AMX Assembler Structures and Constants 373

E. AMX 86 Assembler Interface **381**

Index **Index-1**

AMX 86 USER'S GUIDE
Table of Figures

Page

Section 1: System Description

Figure 2.2-1 AMX General Operation 12
Figure 3.2-1 AMX Task State Diagram 28
Figure 3.9-1 Message Transmission 38
Figure 3.9-2 Message Reception 39
Figure 5.4-1 Simple Time Slicing 80
Figure 5.4-2 Interrupted Time Slicing 80
Figure 5.5-1 Time/Date Format Specification Parameter 89
Figure 12.2-1 Doubly Linked Lists 141

Section 2: System Construction

Figure 14.2-1 AMX Configuration Building Process 165
Figure 14.3-1 Configuration Manager Screen Layout 167

Appendices

Figure C.2-1 User Parameter File 354

1. AMX Overview

1.1 Introduction

The AMX™ Multitasking Executive provides a simple solution to the complexity of real-time multitasking. AMX supervises the orderly execution of a set of application program modules called tasks. AMX allows the system designer to concentrate on the application without becoming overly concerned about the multitasking aspects of the solution.

AMX is based on concepts proven over the past thirty years in minicomputer and microprocessor applications in process control environments. AMX simplifies real-time software implementation by providing the system designer with a well-defined set of rules.

AMX operates in real mode on any Intel® 8086 compatible microprocessor system including the 8086/8088, 80186/188, 80286, Intel386™, Intel486™ and Pentium™. It can also be used on the VAutomation 24-bit Turbo86 processor family. Unless otherwise specified, references in this manual to 8086 infer all of the other processors as well.

AMX gives the system designer complete flexibility and control over the microcomputer configuration employed. A real-time clock must be provided in the configuration if AMX timing facilities are to be employed.

AMX provides a wide variety of services from which the real-time system designer can choose. Many of the services are optional and, if not used, will not even be present in your AMX system. The AMX managers are all optional.

The purpose of this manual is to provide the system designer and applications programmer with the information required to properly configure and implement an AMX-based real-time operating system. It is assumed that you are familiar with the architecture of the processor on which you will be using AMX. It is further assumed that you are familiar with the rudiments of microprocessor programming including the concepts of code, data and stack separation.

For historical reasons, AMX 86 is coded in assembly language. The result is a fast, compact, reliable implementation whose code has remained invariant throughout the lifetime of this very stable product. AMX is provided in source format to ensure that regardless of your development environment, your ability to use and support AMX is uninhibited.

The C programming language, commonly used in real-time systems, is used throughout this manual to illustrate the features of AMX.

Section Summary

This manual is divided into three sections. Each section is divided into chapters.

Section 1 of this manual describes the AMX Multitasking System and how it is used. Separate chapters are provided for each of the AMX managers.

Section 2 describes the AMX System Configuration Builder and the manner in which it is used to create your AMX System Configuration Module.

Section 3 is the application programming guide. It provides detailed descriptions of all of the AMX service procedures which are available in the AMX Library.

AMX Guides

This manual describes the use of AMX 86 for all target processors on which it can be used. Target specific requirements or programming considerations are provided only when necessary to clarify proper operation on a particular processor.

This manual describes the use of AMX in a tool set independent fashion. References to specific assemblers, compilers, librarians, linkers, locators and debuggers are purposely omitted. The **AMX 86 Tool Guide** provides guidance for the proper use of AMX with each toolset with which AMX has been tested.

Guidelines for the proper use of AMX when coding in C are provided in the separate **AMX 86 C Programming Guide**. This guide includes useful debugging tips as well as a description of the AMX Sample Program provided with AMX.

The **AMX 86 Timing Guide** discusses general timing issues related to the use of AMX. Timing metrics generated for specific boards and software development toolsets are also provided. These timing figures can be used as guidelines to expected AMX performance, but are not to be construed as product specifications. The AMX Timing Guide is provided as an appendix in the AMX 86 Tool Guide.

> Note
>
> The AMX 86 PC Supervisor is delivered with AMX for use in the development of applications which must run on PCs with access to PC peripherals and DOS.
>
> The AMX 86 PC Supervisor Reference Manual is provided in its own section of this manual.

1.2 Glossary

Buffer Pool	A collection of data buffers whose use is controlled by the AMX Buffer Manager.
Buffer Pool Id	The handle assigned to a buffer pool by AMX for use as a unique buffer pool identifier.
Circular List	An application data structure used to maintain a list of 1, 2 or 4 byte elements with the ability to add and remove elements at both the top (head) and bottom (tail) of the list.
Clock Handler	The name given to the AMX procedure which is called by the ISP root which services the hardware clock interrupt.
Clock Tick	The interrupt generated by a hardware clock.
Conforming ISP	An Interrupt Service Procedure consisting of an ISP root which calls an Interrupt Handler which has the right to make calls to a subset of the AMX service procedures.
Counting Semaphore	A particular type of AMX semaphore used for event signalling or for controlling access by tasks to multiple resources.
Envelope	The private data storage element used by AMX to pass a message to a task mailbox or message exchange.
Error Code	A series of signed integers used by AMX to indicate error or warning conditions detected by AMX service procedures.
Event Group	A set of 16 events whose access and signalling is controlled by the AMX Event Manager.
Event Group Id	The handle assigned to an event group by AMX for use as a unique event group identifier.
Exit Procedure	An AMX or application procedure executed by AMX during the exit phase when an AMX system is shut down.
Fatal Error	A condition detected by AMX which is considered so abnormal that to proceed might risk catastrophic consequences. Examples include, but are not limited to, insufficient memory in the AMX Data Segment or division by zero in an ISP.
FIFO	First in, first out. Usually used to refer to the ordering of elements in a queue, circular list or linked list.
Group Id	See Event Group Id.
Handle	An identifier assigned by AMX for use by your application to reference a private AMX system data item.

Interrupt Handler — An application procedure called from an ISP root to service an interrupting device.

Interrupt Service Procedure (ISP) — An AMX or application procedure which is executed in response to an external device interrupt request.

ISP — See Interrupt Service Procedure

ISP root — The ISP code fragment (produced by AMX function *ajispm()*) which informs AMX that an interrupt has occurred and calls an application Interrupt Handler.

Kernel Task — The private AMX task which is responsible for all timing control and event sequencing in an AMX system.

Linked List — An application data structure used to maintain a doubly-linked list of arbitrary application data elements with the ability to add and remove elements at head, tail or specified positions in the list.

List Element — An 8-bit, 16-bit or 32-bit value which can be added to or removed from a circular list.

Mailbox — An AMX data structure consisting of a single message queue. Mailboxes allow the interchange of messages between two or more cooperating components (tasks, ISPs, etc.) of an AMX system. Each task or message exchange can have up to four mailboxes.

Memory Block — A portion of a memory section that has been allocated for use by one or more tasks.

Memory Handle — The handle assigned to a private memory section by AMX for use as a unique memory section identifier.

Memory Section — A contiguous region of memory assigned to the AMX Memory Manager for allocation to application tasks.

Message — Application information passed by AMX in an envelope to a task mailbox or message exchange.

Message Exchange — An AMX data structure that consists of four message queues, each for messages of a different priority. The message exchanges allow the interchange and prioritization of messages between two or more cooperating components (tasks, ISPs, etc.) of an AMX system.

Message Exchange Id — The handle assigned to a message exchange by AMX for use as a unique message exchange identifier.

Message Queue	An AMX data structure used to manage messages sent to a task mailbox or message exchange. A separate message queue is provided for each of the four message priorities which a task or message exchange can support.
Message Priority	Identifies which of a task's or message exchange's four message queues is to receive the AMX message.
Nonconforming ISP	An Interrupt Service Procedure which bypasses AMX completely and hence cannot use any AMX service procedures.
RAM	Alterable memory used for data storage and stacks.
Resource Semaphore	A particular type of AMX semaphore used to provide access to an entity such as a math coprocessor, disk file or non-reentrant library whose ownership is to be controlled by the AMX Semaphore Manager.
Restart Procedure	An AMX or application procedure executed by AMX during the initialization phase when an AMX system is started.
ROM	Read only memory of all types including PROMs, EPROMs and EAROMs.
Segment	An area of memory in which AMX code or data is stored. Segments are sometimes called sections or regions according to the nomenclature adopted for a particular processor.
Semaphore	An AMX data structure which can be used by the AMX Semaphore Manager to provide an event signalling mechanism or mutually exclusive access by tasks to specific user facilities.
Semaphore Id	The handle assigned to a semaphore by AMX for use as a unique semaphore identifier.
Slice Interval	The interval of time allocated to a task which is time sliced.
Slot	One of *n* locations used to store list elements in a circular list.
System Configuration Module	A software module, produced by the AMX System Configuration Builder, which defines the characteristics of a particular AMX application.
System Tick	A multiple of the hardware clock tick from which the fundamental AMX unit of time is derived. All time intervals in an AMX system are measured in multiples of the system tick.
Tag	A 4-character name that can be assigned to any AMX system data structure when it is created. A tag can be used to find the identifier of a task, timer, semaphore, event group, message exchange or buffer pool with a particular name.

Task	An application procedure which is executed by AMX in a way which makes it look as though all such procedures are executing at once.
Task Control Block (TCB)	A private data structure maintained by AMX for each task in the system.
Task Id	The handle assigned to a task by AMX for use as a unique task identifier.
Task Priority	The priority at which a task executes. Tasks which have the same task priority are actually ordered in priority according to the order in which the tasks were created.
Task Signal	A set of 15 user-defined event signals associated with each task for task synchronization use.
TCB	See Task Control Block
Time Slice	The process by which AMX allows tasks having the same priority to share the use of the processor in a round robin fashion.
Timer	A facility provided by AMX to permit precise interval measurement in AMX applications.
Timer Id	The handle assigned to a timer by AMX for use as a unique timer Identifier.
Timer Procedure	An application procedure which is executed by the AMX Kernel Task whenever the corresponding timer interval expires.

1.3 AMX Nomenclature

The following nomenclature has been adopted throughout the AMX User's Guide.

Processor registers are referenced as follows:

8-Bit Registers	*AH, AL, BH, BL, CH, CL, DH, DL*
16-Bit Registers	*AX, BX, CX, DX, SP, BP, SI, DI*
Instruction Pointer	*IP*
Flags (Condition Code)	*CC*
Segment Registers	*CS, DS, SS, ES*

The processor flags (condition code) are referenced using the mnemonics employed in software branches as follows:

Zero	*Z*
Non-zero	*NZ*
Carry	*C*
No Carry	*NC*
Positive (no sign)	*NS*
Minus (sign)	*S*
Interrupt flag	*IF*
Direction flag	*UP* if forward; *DN* if backward

Numbers used in this manual are decimal unless otherwise indicated. Hexadecimal numbers are indicated as *0xABCD* in C code or *0ABCDH* in assembly language code.

The terminology *A(Table XYZ)* is used to define addresses. It is read as "the address of Table XYZ".

Addresses are written as segment:offset. For example *ES:BX* is the address with segment determined by extended register *ES* and offset determined by base register *BX*.

Read/write memory is referred to as RAM. Read only memory (non-volatile storage) is referred to as ROM.

AMX symbol names are identified as follows:

ajxxxx	AMX procedure called from C
AAXXXX	AMX procedure called from assembler
AMXXXX	Private AMX procedures, structures and constants
AERXXX	AMX Error Code XXX
AA831xxx.xxx	AMX kernel filenames
AJ831xxx.xxx	AMX C Interface filenames
AMxxxxxx.xxx	AMX reserved filenames
AA832xxx.xxx	AMX PC Supervisor filenames

Throughout this manual examples are provided in C and assembly language. In general, code examples in C are presented in lower case. Code examples in assembler are in upper case. File names are shown in upper case. C code assumes that an *int* is 16 bits as is common for most C compilers for the 8086 family.

This page left blank intentionally.

2. General AMX Operation

2.1 Introduction to Multitasking

A real-time system is characterized by the need to respond rapidly to events occurring asynchronously in time. A multitasking system is one in which a number of activities or processes must be performed simultaneously without interference with each other. A system in which several activities must operate simultaneously with time-critical precision is called a real-time multitasking system.

The AMX Multitasking Executive provides a simple solution to the complexity of real-time multitasking. AMX supervises the orderly execution of a set of application program modules called tasks. Each task solves a particular problem and provides a specific functional capability within the system.

As we all know, the microprocessor can only do one thing at a time. Fortunately, it does all things very quickly. However, to get the effect that all activities are occurring simultaneously, it is necessary to rapidly switch back and forth from one process to another in a well controlled fashion. It is AMX which organizes and controls the use of the microprocessor to achieve this apparent concurrent execution of tasks.

Each task solves a particular problem or provides a specific functional capability within the system. Each task executes independent of other tasks. Facilities are provided, however, to permit tasks to co-operate to achieve a common goal. This process in which more than one task is allowed to share the use of a single processor is called multitasking. The software program which makes it possible is AMX.

What a task does is completely application dependent. In fact, the most difficult aspect of system design is to logically break the problem into a set of tasks which, if implemented, will achieve the desired goal. The following example is presented to illustrate one way in which a simple real-time alarm logging system can be implemented.

Example

Assume that it is necessary to periodically scan a set of digital alarm inputs. When any alarm is detected, a message is to be logged on a printer. The message is to include a description of the alarm and the time of day at which it occurred.

Careful examination of this problem indicates that in fact it is three problems. First, a set of digital inputs must be scanned for the detection of alarms. Second, a message must be printed. Finally, the time of day must be maintained for inclusion in the message.

Three problems usually result in three tasks. Our example is no exception. A time of day task is required to maintain the date and time within the system. The task must be executed once each second if time is to be maintained to the nearest second. We will ignore the requirement to somehow set the proper time and date. (Isn't that another task?)

Alarm scanning will likely be hardware dependent. We will simplify matters by assuming that a scanning task must examine all digital inputs every 100 ms. The task must be capable of detecting alarm changes in the digital inputs. When an alarm is detected, the scanning task must initiate the logging of a message.

However, the scanning task is not allowed the luxury of waiting until the message is printed. It must continue scanning for additional alarms at the same 100 ms rate. The scanning task must, therefore, send to the message task parameters identifying the alarm which has occurred and the time at which it was detected.

The message task is very simple to describe. It receives parameters identifying the time at which a particular alarm occurred. The task must output to a printer a message describing the alarm.

The foregoing example, although very simple, illustrates many of the features of a real-time multitasking system. At first, the implementation of the required set of tasks may be difficult to envision. Two of the tasks must operate periodically at different intervals. The third task, printing, is such a slow process that the other two tasks must be allowed to execute with higher priority. Moreover, provision must be made to allow the alarm parameters to continue piling up on the message task when alarms are occurring at a rate faster than can be printed. (What would we do if we had to print high priority alarms first and delay the printing of lower priority alarms?)

The implementation of the proposed solution is greatly simplified by AMX. AMX will supervise the execution of the three tasks in the defined priority order. Timing facilities are provided by AMX with a resolution governed by the hardware clock. AMX supports message passing between tasks. AMX allows these messages to pile up at the destination task at any of four priority levels because automatic message queuing with priority sorting is an inherent feature of AMX.

This introduction to multitasking is not exhaustive. The intent is to remove some of the mystery from the multitasking concept. The above example is intended to inspire confidence in your ability to understand AMX and use it to solve real-time problems.

2.2 AMX Operation

AMX Startup

Each AMX-based system consists of the AMX executive program and a set of application tasks and interrupt service procedures. This collection of programs resident in the memory of the microprocessor configuration represents the entire operating system.

The manner in which the operating system begins execution is application dependent. In ROM-based systems, automatic hardware vectoring to the program start address is often implemented. In RAM-based systems, the program is first loaded from some storage medium (ROM, hard disk, diskette, etc.) or downloaded from one processor to another. Once the program is loaded, it is started at its start address by the loader.

Figure 2.2-1 illustrates the general operation of an AMX system. Execution begins in the user domain providing the opportunity for hardware specific and application dependent setup prior to the initialization of the AMX system. For example, hardware interfaces may require custom configuring. In some systems, it might be desirable to perform a memory integrity check before system startup is permitted.

Once all custom initialization has been performed, the program calls the AMX entry procedure *ajentr*. Operating characteristics are defined in an AMX System Configuration Module. It is possible to predefine specific tasks and timers which will be automatically created by AMX during its initialization phase. AMX initializes itself and places all application tasks and timers into an idle state.

Once AMX has initialized all of its internal variables and structures, it executes a sequence of user provided Restart Procedures. These procedures can invoke AMX services to start tasks and initialize interval timers.

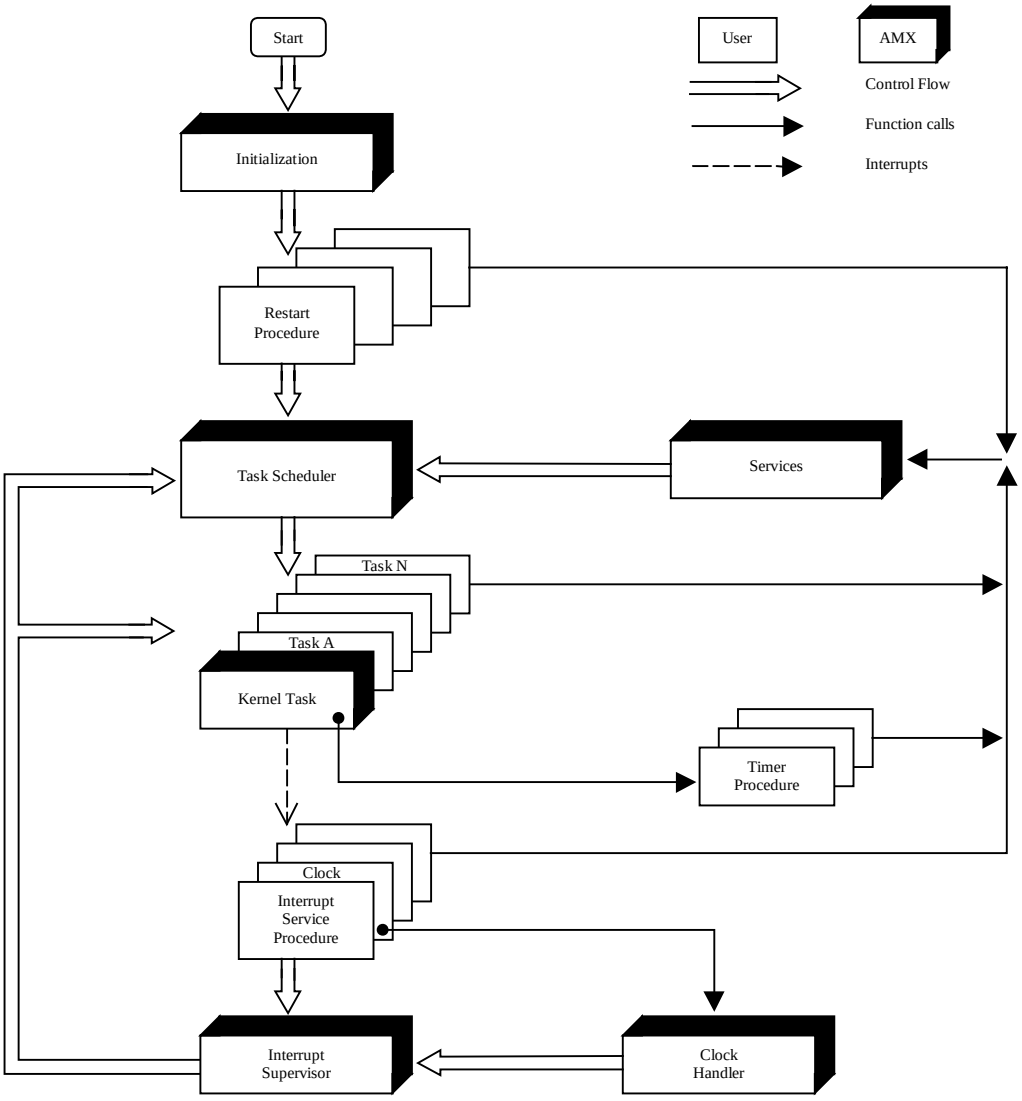


Figure 2.2-1 AMX General Operation

The Task Scheduler

Following system initialization, AMX proceeds to its Task Scheduler. The Task Scheduler searches its list of available tasks to determine the highest priority task capable of execution. Task execution priorities are determined by the system designer. If no task is ready to begin execution, AMX sits with interrupts enabled, waiting for some external event to generate an interrupt.

AMX begins task execution at the task's defined start address. The task executes as though it were the only program in the system. Services offered by AMX can be invoked by the task by procedure calls as indicated in Figure 2.2-1.

Once a task begins execution, it appears to operate without interruption. The interrupts that are periodically taking place are completely hidden from the task by the AMX Interrupt Supervisor and Task Scheduler. The task, once executing, inhibits the performance of all tasks of priority lower than its own. The task continues to execute until it decides to relinquish control, even if only temporarily, by calls to AMX.

The task ends by returning to the AMX Task Scheduler which again finds the next highest priority task ready to execute and gives it control of the processor. A task, once executing, is free to call any of the AMX task services. For instance, a task can send a message to a task mailbox or message exchange, wait for an event or wait for a timed interval. If the task wishes to wait for an event, the AMX service procedure will suspend the task and request the AMX Task Scheduler to force execution of the next highest priority task ready for execution.

AMX acts as the context switcher supervising the orderly execution of application tasks. AMX employs a preemptive, priority-driven scheduling algorithm which ensures that the highest priority task which is ready to do useful work always has control of the processor.

AMX will switch tasks if it receives a request from the executing task to perform an operation which, of necessity, invokes a task of higher priority. For instance, the executing task may request AMX to start a higher priority task.

The Interrupt Supervisor

Tasks execute with the processor interrupt facility enabled to permit service of external devices. When an external interrupt occurs, the task is interrupted in the manner dictated by the processor. The processor automatically saves the return address and some subset of the processor state (registers, flags, etc.) and branches to an Interrupt Service Procedure (ISP). The exact vectoring method is determined by the hardware configuration employed in the system.

Two types of ISPs exist: nonconforming ISPs and conforming ISPs.

A **nonconforming ISP** must quickly service the device to remove the interrupting condition. The ISP must preserve all registers which it uses. The nonconforming ISP cannot make calls to any AMX service procedures.

A **conforming ISP** can make use of a subset of the AMX service procedures. A conforming ISP consists of an ISP root and an Interrupt Handler. The processor vectors to the ISP root which informs the AMX Interrupt Supervisor that the interrupt has occurred. The Interrupt Supervisor preserves the state of the interrupted task and, if necessary, switches to an interrupt stack. The Interrupt Supervisor then calls the associated Interrupt Handler.

The Interrupt Handler must quickly service the device to remove the interrupting condition. The handler is free to make procedure calls to a subset of the AMX service facilities. When device service is completed, the AMX Interrupt Supervisor dismisses the interrupt.

The AMX Interrupt Supervisor monitors calls made by the Interrupt Handler to AMX service procedures. If no such calls have been made, AMX automatically restores the state of the interrupted task and returns directly to the interrupted task at its point of interruption.

The Interrupt Handler may have requested AMX to initiate or resume execution of some task of higher priority than the interrupted task. If so, the AMX Interrupt Supervisor suspends the interrupted task and marks it as ready to resume execution at the earliest opportunity. The AMX Task Scheduler is then invoked to determine the highest priority task capable of execution.

The AMX Interrupt Supervisor supports nested interrupts on processors which provide this capability. If interrupts nest, the Interrupt Supervisor defers its task switching checks until all of the concurrent interrupts have been serviced.

> Note
>
> A conforming ISP root can be created using AMX procedure *ajispm*.

Timing Facilities

The AMX Timer Manager provides a Clock Handler and a Kernel Task to provide complete timing control for your real-time application. The AMX Clock Handler is independent of any particular hardware configuration. If AMX timing facilities are to be utilized, a real-time clock must be included in the configuration.

The hardware clock interrupt must be serviced by a conforming ISP. Whenever a clock interrupt occurs, the application Interrupt Service Procedure must dismiss the hardware clock interrupt and call the AMX Clock Handler.

The AMX Clock Handler triggers the AMX Kernel Task if required. The Kernel Task is triggered at the user defined system tick interval if, and only if, there is any outstanding timing activity required in the system. In this case, the interrupted task is suspended and the AMX Kernel Task begins execution.

The AMX Kernel Task executes as the highest priority task in the system. The AMX Kernel Task monitors all tasks which are in a timed wait state. If a task's timer expires, the AMX Kernel Task primes the task to resume execution with a timeout indication.

The AMX Kernel Task also services all expiring application interval timers. Whenever an application interval timer expires, the corresponding application Timer Procedure is executed. This procedure can invoke a subset of the AMX services to send messages, signal events or wake tasks. If the timer is defined to be periodic, the AMX Kernel Task automatically restarts it with the predefined period.

Message Queuing

One of the more powerful features of AMX is its ability to queue messages for tasks. The queuing facility permits messages to pile up in a controlled fashion, freeing the ISP, Timer Procedure or task which is sending the message to continue with its appointed function. If a task sends a message, it can suspend itself until the message has been received and processed by some other task.

The AMX message queuing facility is further enhanced by allowing the messages to queue at any of four priority levels. A task can therefore receive messages from a variety of callers with the messages already sorted in order of priority by AMX.

The system designer describes the exact message queuing requirements in a task definition or in a message exchange definition. For each task and message exchange, you can specify which, if any, of the four message queuing priority levels are to be supported. You also specify the required message nesting depth for each of these message queues, often referred to as mailboxes.

AMX maintains a free list of message envelopes. These envelopes are used by AMX to transmit messages to the mailboxes of tasks and message exchanges. The caller's message parameters are moved into a free envelope which is then added to the message queue of the destination task mailbox or message exchange mailbox.

The AMX message queuing facility is especially useful in event logging applications. In such applications, messages are transmitted to a printing task by any task wishing to log an event. The printing task is then executed once for each unique request which it has received. High priority messages can easily be forced to precede low priority messages using the AMX message priority feature. Finally, any task wishing to wait until its particular message has been logged can do so by informing AMX of its intent at the time the message is sent.

AMX Shutdown

An AMX system can be shut down in the same orderly fashion in which it was started. A task initiates a shutdown with a call to the AMX exit procedure `ajexit`.

It is the caller's responsibility to assure that the AMX system is in a reasonable state to permit a shutdown. For instance, all device I/O operations should be stopped and all timing activity should be curtailed.

The AMX shutdown procedure executes a series of user Exit Procedures which can restore the original environment which existed prior to starting the AMX system. Device interfaces can be reprogrammed and interrupt vectors restored if necessary.

Once all of the Exit Procedures have been executed, AMX returns to the point at which AMX was originally launched.

2.3 AMX Managers

AMX provides a set of managers to simplify event synchronization, resource manipulation and memory allocation. Not all applications will make use of all of the managers. The system designer can decide which of the AMX managers is best suited for a particular application.

The **Time/Date Manager** provides Y2K compliant time of day calendar support if required. The AMX calendar clock includes second, minute, hour, day, month, year and day of the week. AMX services are provided to set and read the calendar clock. A formatting procedure is also provided to translate the calendar time and date from the internal format in which it is maintained by AMX into an ASCII string in several of the most popular formats.

An application procedure can be tied to the calendar clock and called at one second intervals to permit simple time of day event scheduling.

The **Semaphore Manager** provides two types of semaphores each with priority queuing and timeout: resource semaphores and counting semaphores.

A resource semaphore can be used to provide controlled access to your resources. It uses a binary semaphore to limit access to each resource to one task at a time. Ownership and release of a resource is governed by calls to the Semaphore Manager. A resource semaphore offers the unique characteristic of identifying each resource owner. Only the task which owns a resource is permitted to release it.

General purpose counting semaphores can be created for mutual exclusion and resource management. Tasks must request the Semaphore Manager for access to the resource controlled by such a semaphore. The task can specify the priority of its request to use the semaphore. If the semaphore is not free, the task will be forced to wait for its availability. The task will be placed on the semaphore wait queue at the priority specified by the task. Optionally, the task can specify a timeout interval limiting the time the task is prepared to wait.

A task, ISP or Timer Procedure can signal the semaphore with a call to the Semaphore Manager. The Semaphore Manager grants access to the semaphore to the task, if any, waiting at the top of the semaphore's wait queue.

The **Event Manager** provides a convenient method for synchronizing one or more tasks to events detected in Interrupt Service Procedures, Timer Procedures and other tasks. A task requests the Event Manager to suspend its operation until any one of a particular set of events occurs. Alternatively, the task can request to wait until all of a set of event conditions are met. Optionally, the task can specify a timeout interval limiting the time the task is prepared to wait. More than one task can be waiting for the same event or set of events.

When a task, ISP or Timer Procedure detects an event, it signals the event with a call to the Event Manager. The Event Manager checks to see if the event has resulted in an event combination for which one or more tasks are waiting. If so, the tasks which were waiting are allowed to resume execution.

The AMX 86 **Task Mailbox** facility provides a general purpose message queuing mechanism for tasks. This service is not provided by a separate AMX manager; it is an inherent feature of AMX 86. Any task can have up to four private mailboxes in which the task can receive AMX messages. Tasks, ISPs or Timer Procedures can send messages to such a task mailbox. The messages are ordered in each task mailbox according to their order of arrival. If such a task is idle when a message is deposited in one of its mailboxes, the task will automatically be started and given the message extracted from the mailbox. At any time, a task can request a message from any of its mailboxes. When the task ends, it will automatically be restarted if its mailboxes are not all empty. The task will be given the highest priority message available.

The **Message Exchange Manager** provides a general purpose prioritized message queuing facility. Tasks, ISPs or Timer Procedures can send messages to a message exchange to be queued at any of four priority levels. When a task requests a message from a message exchange, it is given the highest priority message available. If no message is available, the task will be forced to wait. The task will be placed on the message exchange wait queue at the wait priority specified by the task. Optionally, the task can specify a timeout interval limiting the time the task is prepared to wait. This interval can be from no wait to an indefinite wait. When a message subsequently arrives, it is immediately given to the task waiting at the top of the message exchange's wait queue.

The **Buffer Manager** provides fast, efficient access to multiple pools of buffers, each buffer representing a fixed size block of memory. This form of memory management meets the requirements of most typical applications and is best suited for real-time use in which memory block availability must be predictable and in which the penalties for memory fragmentation cannot be tolerated.

Application modules can request the Buffer Manager to get a buffer from a pool. Unlike other AMX managers, the Buffer Manager does not permit a task to wait for a buffer to become available.

When released, the buffer is automatically returned by the Buffer Manager to the pool to which the buffer belongs. Buffer ownership can be increased so that more than one task can simultaneously own a shared buffer. Special facilities are provided to assure that if a buffer is owned by more than one task, it is only returned to its pool when the slowest owner finally releases it.

The **Memory Manager** controls the dynamic allocation of memory to tasks in the multitasking environment. Multiple sections of user defined memory can be controlled by the Memory Manager. A section can exceed 64K bytes. The memory in each section must be contiguous but the sections themselves do not have to be contiguous.

A task can request the Memory Manager to allocate a contiguous block of memory of any size. When finished with the block, the task requests the Memory Manager to free the memory for use by other tasks.

A particularly unique feature of the Memory Manager permits any block of memory (including those acquired from the Memory Manager) to be treated as memory from which smaller private blocks can be dynamically allocated.

The **Circular List Manager** provides a general purpose circular list facility for maintaining compact lists of 8-bit, 16-bit or 32-bit variables. Circular lists are particularly useful for managing character streams associated with input/output devices.

The **Linked List Manager** provides a fast, general purpose doubly-linked list facility for maintaining lists of arbitrary application data structures (objects).

The Linked List Manager removes the tedium and the frequent errors usually encountered when each application must manipulate the linkages of different types of objects on different lists. Objects can reside on multiple lists at the same time, a characteristic frequently encountered in real problems but ignored by most list manipulation software.

2.4 Starting AMX

An AMX operating system consists of AMX, the subset of its managers which you choose to use and your complement of application programs. All of these modules are connected together to form the AMX operating system as described in the AMX Tool Guide.

Before launching AMX, you must establish the required operating mode for the particular target processor you are using. AMX 86 operates only in the 8086 real mode.

AMX uses the Medium or Large segmentation model in which multiple code segments and one or more read/write data segments are provided. AMX makes no assumption about the manner in which these segments correlate to physical memory.

AMX is launched with a launch parameter defining the nature of the launch. The launch parameter is a 16-bit unsigned integer in which each bit defines a particular launch characteristic. Bit mask mnemonics *AMLPxx* are defined in include files *AMX831SD.H* and *AMX831SD.DEF*.

Bit	Value	Launch parameter
0	*0*	Launch is permanent
	AMLPTMP	Launch is temporary
1	*0*	Vector Table is not alterable (in ROM or not accessible)
	AMLPVA	Vector Table is alterable (in RAM and accessible)
2	*0*	Interrupts disabled during launch
	AMLPIE	Interrupts enabled during launch

AMX is always launched from your *main* program (or startup module) by calling AMX procedure *ajentr* from C or *AAENTR* from assembly language.

Your AMX operating system can be launched in two ways: permanently or temporarily. The type of launch is determined by bit 0 of the launch parameter.

Bit 1 of the launch parameter indicates whether or not AMX will be permitted to alter the processor vector table. For most applications, the vector table is in RAM and therefore alterable.

AMX disables the interrupt system at the time you launch AMX. Unless bit 2 of your launch parameter indicates otherwise, interrupts will remain disabled during the startup process while AMX initializes its internal parameters and calls each of your Restart Procedures.

Permanent Launch

In most applications, your AMX operating system is resident in ROM or loaded into RAM. AMX is started in real mode and given permanent control of the processor.

An AMX system can be launched permanently from a main program coded in C as illustrated in the following example.

The implication of starting AMX from a main C program is that a C startup module has already provided a stack before procedure *main* was called. The *main* procedure calls AMX at its entry point *ajentr* with three parameters. The first parameter is the launch parameter. The value *AMLPVA* indicates that the launch is permanent (bit 0 is 0), vectors are alterable (bit 1 set by *AMLPVA*) and interrupts are to be disabled during the launch (bit 2 is 0).

The second parameter is the pointer to your application's User Parameter Table in the System Configuration Module which defines the characteristics of your AMX system. The third parameter, *NULL*, must be present but is not used by AMX because the launch is permanent.

```
/* Start AMX for permanent execution                              */

#include "amx831sd.h"                    /* AMX Structure Definitions  */


void main()
{
            struct amxupts FAR *uptp;    /* User Parameter Table pointer*/

            :
            ajupt(&uptp);                /* Fetch pointer to UPT       */
                                         /* Start AMX: no exit,        */
                                         /* vectors alterable and      */
                                         /* interrupts disabled        */
            ajentr(AMLPVA, uptp, NULL);
            }
```

If the vector table is in ROM or, for any reason, is not to be altered by AMX, set the launch parameter to 0 instead of *AMLPVA*. In this case, you will not be able to use AMX services to dynamically install pointers to Interrupt Service Procedures into the vector table.

If you wish interrupts enabled during the launch process, add *AMLPIE* to the launch parameter. Be sure that interrupts from a particular device are inhibited until you have installed an ISP to handle interrupts from the device.

An AMX system can be launched permanently from a startup module coded in assembly language as illustrated in the following example.

Systems of this type begin execution at the AMX entry point *AAENTR* with the launch parameter in register *BX*. Register *BX[0]* must be set to zero to indicate that a permanent launch is required. Register pair *ES:SI* must provide a pointer to the User Parameter Table in the System Configuration Module which defines the characteristics of your AMX system.

Set register *BX[1]* to 1 (*AMLPVA*) if interrupt vectors are to be alterable. If you set register *BX[1]* to 0, you will not be able to use AMX services to dynamically install pointers to Interrupt Service Procedures into the vector table.

Set register *BX[1]* to 0 if interrupts are to be disabled during the startup process. If you set register *BX[1]* to 1 (*AMLPIE*) to enable interrupts during the launch, be sure that interrupts from a particular device are inhibited until you have installed an ISP to handle interrupts from the device.

When AMX is started in this manner, it immediately establishes the AMX Kernel Stack as the current stack. There is therefore no need for your application to provide a stack for a permanent launch.

In a ROM based system, AMX is frequently restarted whenever the 8086 processor is reset. A startup module, coded in assembler, receives control, initializes the 8086, enters protected mode and starts AMX as illustrated in the following example.

```
          EXTRN     AAENTR:FAR          ;AMX Entry Point
          EXTRN     _AMXUPT_:DWORD      ;A(User Parameter Table)
;
USER_CODE SEGMENT BYTE 'CODE'
;
          ASSUME    CS:USER_CODE
;
UPTADR    DD        _AMXUPT_            ;A(User Parameter Table)
;
START     PROC      FAR
          :
          Initialize 8086
          :
          LDS       SI,CS:UPTADR        ;DS:SI = A(User Parameter Table)
          MOV       BX,AMLPVA           ;BX = no exit, vectors alterable
                                        ;and interrupts disabled
          JMP       AAENTR              ;start AMX
;
START     ENDP
;
USER_CODE ENDS
```

Temporary Launch

Your AMX operating system can be started, allowed to run for a while and then stopped. This type of operation is called a temporary launch. The most common application of this type occurs on PC compatibles. An AMX operating system is started from DOS, allowed to operate for a while and then forced to return to DOS.

If you wish to start an AMX system for temporary execution from a module coded in assembler, refer to the source code for procedure *ajentr* in AMX source file *AJ831KI.ASM* in installation directory *AMX831\CIF*. Mimic the simple operations which are provided by that procedure.

An AMX operating system can be started for temporary execution from a main C program as in the following example.

AMX is started at its entry point *ajentr* with three parameters. The first is the launch parameter *AMLPTMP+AMLPVA* indicating that the launch is temporary (bit 0 set by *AMLPTMP*), vectors are alterable (bit 1 set by *AMLPVA*) and interrupts are to be disabled during the launch (bit 2 is 0).

If the vector table is in ROM or, for any reason, is not to be altered by AMX, replace the value *AMLPVA* in the launch parameter with the value 0. In this case, you will not be able to use AMX services to dynamically install pointers to Interrupt Service Procedures into the vector table.

If you wish interrupts enabled during the launch process, add *AMLPIE* to the launch parameter. Be sure that interrupts from a particular device are inhibited until you have installed an ISP to handle interrupts from the device.

As for a permanent launch, the second parameter is a pointer to your User Parameter Table. The third parameter is a pointer to a double word variable of your choice.

When an AMX system is launched for temporary execution, it executes until one of your application tasks calls the AMX exit procedure *ajexit* requesting an orderly shutdown of the AMX system (see Chapter 3.11). The *ajexit* caller can return two parameters to the procedure that launched AMX. One of these parameters is an integer which is received as *errcode* from the *ajentr* call. The other is a double word parameter stored in the variable provided in the call to *ajentr*.

In the following example, the integer result is stored in variable *errcode* and the double word parameter is recorded in variable *resultp*.

```
/* Start AMX for temporary execution                                  */

#include "amx831sd.h"              /* AMX Structure Definitions       */

void main()
{
      struct amxupts FAR *uptp;    /* User Parameter Table pointer    */
      int      errcode;
      char     *resultp;
      :
      :
      ajupt(&uptp);                /* Fetch pointer to UPT            */
                                   /* Start AMX: exit allowed,        */
                                   /* vectors alterable and           */
                                   /* interrupts disabled
      errcode = ajentr(AMLPTMP+AMLPVA, uptp, &resultp);

      :
      Interpret your termination status (errcode) and, if necessary,
      look at your results referenced by the pointer returned in
      pointer variable resultp.
      :
      }
```

3. Application Tasks

3.1 Task Creation

The AMX Multitasking Executive provides a simple solution to the complexity of real-time multitasking. AMX supervises the orderly execution of a set of application program modules called tasks. Each task solves a particular problem and provides a specific functional capability within the system.

The maximum number of tasks in a system is user defined in your System Configuration Module (see Chapter 14.5). The defined maximum sets an upper limit on the number of actual tasks that can be created by the application. AMX 86 sets an upper bound of 100 tasks in a system. Other versions of AMX have no such restriction.

Tasks can be created in two ways. Tasks can be predefined in your AMX System Configuration Module which is processed by AMX at startup. Tasks defined in this fashion are automatically created by AMX but are not started. Restart Procedures and tasks can also dynamically create tasks using AMX procedure *ajtkcre*.

AMX assigns a task identifier (id) to each task when it is created. The task id is a handle which uniquely identifies the particular task. When AMX prebuilds a task, it saves the task identifier in the id variable specified in the task's description.

When tasks are created dynamically, AMX returns the assigned task id to the creator. It is the responsibility of the application to keep track of the task id for future reference to the task.

When a task is created, you can provide a unique 4-character task tag to identify the task. The tag can be used subsequently in a call to *ajtktag* to find the task id allocated by AMX to the particular task.

AMX uses the information in the task definition to construct a Task Control Block (TCB) for the task. The TCB of each task is used exclusively by AMX to control task execution. At any instant in time, the content of the TCB, as maintained by AMX, completely describes the state of the corresponding task.

Tasks which do not receive messages are written as Large or Medium model C functions without formal parameters. These tasks must be started using AMX procedure *ajtrig*. For this reason, such tasks are called **trigger tasks**. For example, a task that immediately ends would appear as follows:

```
void cdecl task1(void)
{
   }
```

Tasks which must receive a message are written as Large or Medium model C functions which receive a set of parameters. Tasks that receive messages are called **message tasks**. These tasks must be started using any of the variations of AMX procedure *ajsend* or *ajsenw*. For example, a task which receives an integer as a message and then ends would appear as follows:

```
void task2(int message)
{
   }
```

A message task must be defined to include one to four message queues (also called task mailboxes) in which the task receives messages. Messages which arrive in the task's mailboxes are automatically delivered to the task on the task's stack ready for processing by the task. This type of message processing is described in Chapter 3.9.

Tasks written in assembly language must be coded as *FAR* procedures as follows:

```
TASK_CODE SEGMENT BYTE 'CODE'
;
; The task is located in user program memory
;
         ASSUME   CS:TASK_CODE
;
STTASK   PROC     FAR
         :
         Set DS and ES if required by task
         Access parameter on stack via BP
         Do task functions
         :
         RET                              ;Return to AMX
;
STTASK   ENDP
;
TASK_CODE ENDS
```

3.2 Task States

A task is always in one of the following states:

Idle
Ready
Run
Wait
Halt

When a task is created, AMX assigns it a Task Control Block and sets it in the idle state. An idle task has no outstanding requests to execute pending. It is waiting to be triggered.

A ready task has an outstanding request to execute or is ready to resume execution after having been interrupted or waiting.

A task which is executing is the only task which is in the run state.

A task is in the wait state when it is blocked pending the occurrence of some event. The wait state is always qualified with an indication in the TCB as to what the task is waiting for.

The halt state is reserved for use by KADAK debug utilities to suspend all task execution for debugging purposes.

The halt state is used by the AMX Breakpoint Manager to suspend all task execution when a debug breakpoint is encountered. When a debug breakpoint is encountered, all application tasks except the one in which the breakpoint occurred are placed in the halt state. When you leave the debugger, the tasks are restored to the state they were in prior to the breakpoint.

Figure 3.2-1 illustrates the task states and shows the state transitions which are possible. The halt state is not explicitly shown.

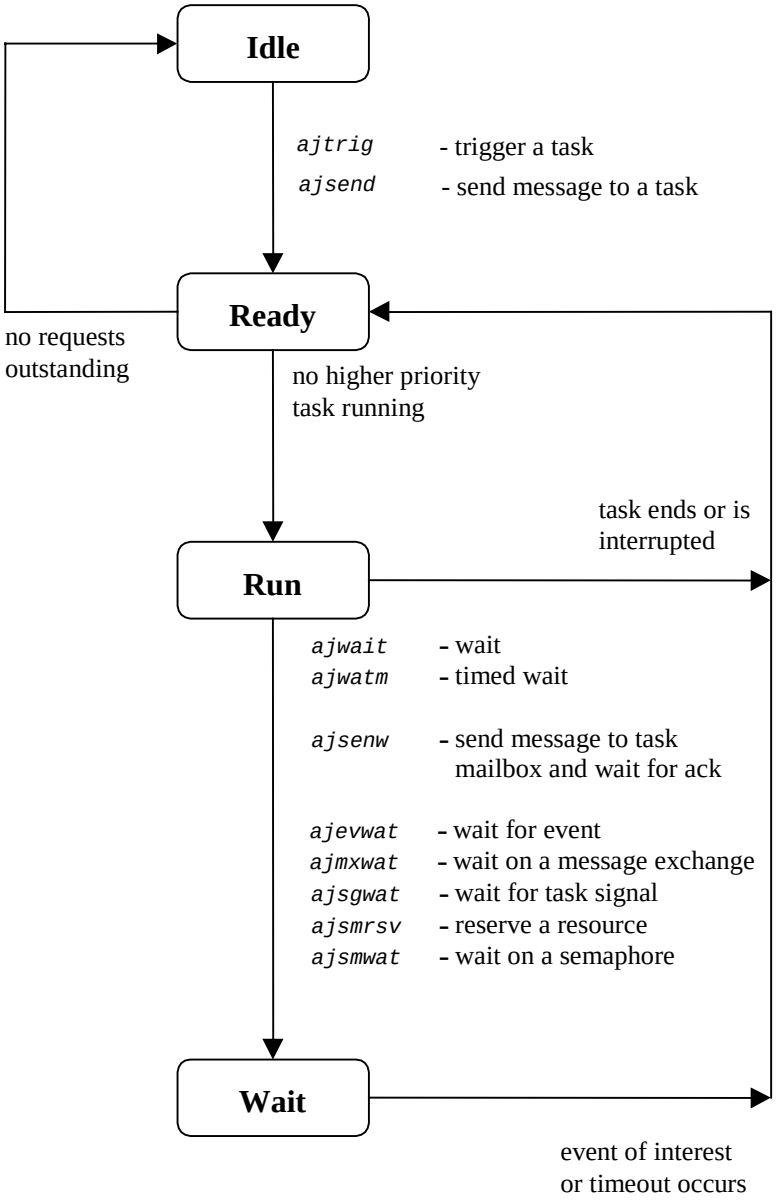


Figure 3.2-1 AMX Task State Diagram

3.3 Starting a Task

At startup, AMX initializes all predefined application tasks into an idle state. Once idle, a task cannot execute until AMX receives a directive to start the task. How then does an AMX system get off the ground?

Three AMX directives are provided to start a task. These directives are procedure calls to AMX. The *ajtrig* call is used to trigger a task without a message. The *ajsend* call is used to send a message to a task. The *ajsenw* call is used to send a message to a task and wait for it to be received and acted upon. In response to any of these calls, AMX primes the task for subsequent execution. In the case of *ajsend* or *ajsenw* calls, the caller's message parameters are moved into a message envelope. The envelope is then inserted in the destination task's mailbox at the priority level indicated by the caller.

Requests to start a task can be issued in any of the following application modules.

Restart Procedure
Application Task
Interrupt Service Procedure
Timer Procedure

The Restart Procedure provides the first opportunity to start an application task. At startup, after all predefined tasks have been created, AMX executes all of the Restart Procedures described in the Restart Procedure List. A Restart Procedure can be used to request execution of one or more tasks in the system. The AMX Task Scheduler subsequently starts the highest priority task capable of execution.

Once a task is executing, it is allowed to request the execution of any task in the system, including itself. A task can use the *ajsend* or *ajsenw* call to send a message to another task. The *ajsenw* call is only available to tasks. A task issuing an *ajsenw* call to AMX suspends itself (waits) until such time as the called task has executed in response to the call. AMX guarantees that the called task is executed in response to one caller at a time.

Requests for task execution can also be initiated in response to device interrupts. When an interrupt occurs, the device Interrupt Service Procedure (ISP) is executed. The ISP can issue an *ajtrig* call to start a particular task. It can issue an *ajsend* call if a message must be sent to the task being called. Whenever an ISP requests to start a task, AMX temporarily suspends the interrupted task, thereby leaving it in a state ready to resume execution. The AMX Task Scheduler is then invoked to start (or return to) the highest priority task ready for execution. It is this type of operation that permits a task to begin execution in response to an event which is considered to be of higher priority than the task which was running.

AMX also permits tasks to be executed at periodic intervals. For example, a periodic application interval timer (see Chapter 5) can be started by your application. When the timer expires, the AMX Kernel Task executes the associated application Timer Procedure. The Timer Procedure can issue an *ajtrig* or *ajsend* call to AMX to request execution of any given task.

3.4 Task Priority

Task priorities are used by the AMX Task Scheduler to determine which task to execute. At all times, the task with the highest priority which is capable of execution will run.

A task's priority is defined at the time the task is created. Task priorities range from 0 (highest) to 255 (lowest). Priorities 0 and 128 to 255 inclusive are reserved for AMX use. The AMX Kernel Task executes at priority 0 above all other tasks.

Application tasks can be assigned priorities 1 to 127 inclusive. Since the number of priority levels exceeds the maximum number of possible tasks, it is evident that priority levels do not have to be shared.

If more than one task is assigned the same priority, AMX will assign the tasks relative priorities according to the chronological order in which they were created with the first created task having the higher priority.

A task may change its priority or the priority of any other task. This practice is not recommended however. Experience has shown that this facility is rarely required and all too often is abused. The task's new priority remains in effect until you decide, if ever, to change it again.

3.5 Task Execution

AMX starts a task by making a *FAR* procedure call to the task. AMX starts execution of a task at the task start address specified in the task's definition. The task is started in response to a request for its execution. Requests can come from Restart Procedures, tasks, Interrupt Service Procedures or Timer Procedures (see Chapter 3.3).

AMX starts the task when no tasks of higher priority are capable of execution. A task can therefore only execute if all higher priority tasks are idle or suspended for some reason.

When AMX starts the task, the following conditions exist:

Interrupts are enabled.
All registers are free for use.
DS,ES *DGROUP* segment
SS:SP Task stack ready for use
BP Offset of received message on stack
The direction flag is set to forward.

Tasks started in this manner can be readily coded in C or assembler.

A message may have been passed to the task. Message passing is described in more detail in Chapter 3.9. A task receives message parameters on its stack. The processor base pointer register (*BP*) is initialized by AMX to the offset of the first byte of a message on the task stack.

A caller's message parameters are moved into a message envelope by AMX for transmission to a task. When the task starts, the parameters in the message envelope are transferred to the task's stack and the envelope is automatically released for reuse.

The number of parameter bytes that can be passed to a task is configured by you in the AMX User Parameter Table. The content of the message is completely user defined.

Once a task begins execution, it has complete control of the processor. The interrupt system must be left enabled to permit device interrupts to be serviced. If a task must disable interrupts for some reason, it is recommended that this period be kept as short as possible so that the system's interrupt response time is not degraded.

The application task can execute without concern for the fact that interrupts will occur and will be serviced. If higher priority tasks become ready for execution, the task will be suspended temporarily by AMX. When higher priority tasks become suspended or have completed their operation, the interrupted task will be permitted by AMX to resume execution from its point of interruption.

Occasionally a task must perform a sequence of operations without interference by other tasks. If the sequence is too long to permit interrupts to be disabled, the task can request AMX to become temporarily privileged. During the period of privileged execution, interrupts remain enabled but execution of any higher priority task, including the AMX Kernel Task, is inhibited. Privileged operations should be kept as short as possible.

3.6 Task and Event Synchronization

AMX offers several simple forms of task/event synchronization.

Using the *ajwait* call, a task can unconditionally **wait** for an event. The event can be task dependent, device dependent or time dependent. The task, having issued an *ajwait* call, remains suspended unconditionally until another task, an Interrupt Service Procedure or a Timer Procedure issues an *ajwake* call requesting AMX to **wake** up the particular waiting task. This *ajwait/ajwake* pair can be used to provide simple event synchronization.

AMX also supports this form of synchronization with an automatic timeout facility. A task can issue an *ajwatm* call specifying the maximum interval which the task is willing to wait for the event to occur. If no other task, ISP or Timer Procedure issues an *ajwake* call in the interim, AMX will automatically wake up the task when the interval expires. The task receives from AMX an indication whether or not a timeout occurred.

A more powerful and flexible form of task/event synchronization is supported by **task signals**. Each task owns a set of 15 user defined task signals. A task can issue an *ajsgwat* call to wait, with optional timeout, for a specific combination of signals to be set.

Tasks, ISPs and Timer Procedures can call AMX procedure *ajsgnl* to set any of a particular task's task signals when corresponding events are detected. The task receives an indication whether or not a timeout occurred.

A task can also be synchronized to another task using the AMX **call and wait** message passing facilities. A task sends a message to another task using the AMX *ajsenw* call which forces the sender to wait. When the receiving task eventually receives the sender's message, it can issue an *ajwakc* call to wake the calling task. The sender will then resume execution knowing full well that its message has already been received by the task to which it was sent. If the receiving task does not issue an *ajwakc* call, AMX will do so automatically when the receiving task ends. No timeout is provided with this type of synchronization.

These synchronization facilities offer the advantage of simplicity. These mechanisms are an inherent part of the AMX kernel. The disadvantage of each is that the event signaller must know the task id of the task which is waiting or to whom a message must be sent.

The simple features included in the AMX kernel are augmented by three powerful mechanisms for event synchronization provided by separate AMX managers.

The **Semaphore Manager** provides counting semaphores with queuing and timeout facilities for mutual exclusion and resource management. It also offers a unique resource semaphore which extends to semaphores the concept of ownership of the corresponding resource.

The **Event Manager** offers the best solution for complex event coordination. It permits a task to be synchronized to any or all of a particular set of events. It also allows more than one task to wait for the same events.

The **Message Exchange Manager** allows a task to be synchronized to another task using its call and wait message passing facilities. It allows one or more tasks to wait at a message exchange for the arrival of a message. Tasks, ISPs and Timer Procedures can send messages to the message exchange using the AMX *ajmxsnd* call. The receiving task must use the AMX procedure *ajmxwat* to wait for a message at the message exchange.

Each of the synchronization methods provided by the managers share several common features. In each case, the signaller does not need to know the identity of task(s) waiting for its signal. When multiple tasks wait on a semaphore, event group or message exchange, each task specifies the priority at which it wishes to wait. Finally, any task waiting on a semaphore, event group or message exchange can specify the maximum interval which the task is willing to wait.

3.7 Task Timing

The AMX Clock Handler and Kernel Task act as a Timer Manager providing timing facilities for use by tasks. It has been shown in Chapter 3.6 that tasks can wait for an event to occur with an automatic timeout. The task is suspended following its wait request until the event occurs or the interval specified in the call expires.

The *ajwatm* call to AMX can also be used by a task to implement a delay. The delay interval is specified in system ticks. The task is suspended until the interval expires. AMX assures that the task automatically resumes execution, provided that no higher priority task is able to execute.

Other timing functions required by the task can be implemented using interval timers (see Chapter 5). Timers are 32-bit counters. Timing resolution is in multiples of the system tick. The AMX routine *ajtmcnv* is available to convert milliseconds to the equivalent number of system ticks.

Timers can be created at any time by a call to the AMX routine *ajtmcre*.

A timer is started by writing the timer period to it using the AMX routine *ajtmwr*. At any instant, a task can read the time remaining in the interval by calling AMX routine *ajtmrd*. A timer can be stopped by writing zero to it. A timer can be deleted when it is no longer required by a call to *ajtmdel*. These simple procedures give the task complete control over interval timing.

Whenever an interval timer expires, AMX executes an associated Timer Procedure. Using this feature, a task can start an interval timer and rest assured that, when the interval expires, the action determined by the associated Timer Procedure will be performed. Since the Timer Procedure is called by the AMX Kernel Task which has the highest priority in the system, the Timer Procedure executes at a priority higher than that of any application task.

Interval timers must be used by tasks wishing to measure time. Instruction counting loops are of no value in a multitasking system. Since a task is being constantly interrupted and occasionally suspended to execute higher priority tasks, any timing performed by counting instructions within a program loop will be in error.

3.8 Ending a Task

When a task completes its appointed function, it must relinquish control of the processor to AMX. The AMX Task Scheduler will then give control of the processor to the next highest priority task ready to execute.

AMX starts a task by a *FAR* procedure call to the task at the task start address. The task program is, therefore, a procedure. When the task is finished, it returns to AMX in normal end-of-procedure fashion.

A task may wish to abort execution under error conditions. The task's stack may be deeply nested when the error condition arises. To allow a task to terminate under these circumstances, AMX provides the *ajend* exit facility. Any task which calls *ajend* is immediately terminated by AMX.

When a task is triggered or a message is sent to a task, the task receives a request to execute. A task executes once for each such request. When the task ends, AMX examines the task to determine if any additional requests for execution of the task are pending. If outstanding requests are present, AMX flags the task as ready to run again. The AMX Task Scheduler will then immediately start this task again. This process continues until the task finally ends and no requests for execution of the task are present. At that time, AMX places the task into the idle state.

If the task has received a message, AMX must perform some additional processing when the task ends. AMX checks to see if the message came from a task. If so, AMX tests to see if the sender is waiting for the termination of the task just ending. If the sender is waiting, AMX sets it into the ready state ready to resume execution.

It is possible that an application task may never end. Such a task, once started, runs forever. For example, a task might wait for an event and then do some event-dependent processing. Once the processing is complete, the task waits for the next event.

In this example, the task never reaches a logical end. Note, however, that the task does become suspended awaiting an event. A task which has no logical end and which never suspends itself is said to be compute bound.

Even if a task does wait, it is still possible that the task may effectively be compute bound. For instance, assume that a low priority task repetitively sends a message to a higher priority task and waits for an answer. Also assume that the higher priority task will always provide an immediate response. In this case, the lower priority task will always be allowed to resume after its message is sent even though a temporary suspension does occur. The task will block all lower priority tasks.

> Note
>
> A compute bound task inhibits execution of all lower priority tasks.

3.9 Message Passing

AMX supports the passing of a message to a task mailbox or message exchange.

Messages can be sent to a task by:

Application Tasks
Interrupt Service Procedures
Timer Procedures
Restart Procedures
Exit Procedures

You can send a message to a task mailbox or message exchange using the AMX procedure *ajsend* (or its variation *ajsendp*) or *ajmxsnd* respectively. The maximum number of parameter bytes that can be sent in a message is configured by you in your System Configuration Module. The parameter bytes are completely application dependent.

Messages must be integer aligned. In the following examples, it is assumed that your C compiler aligns arrays and structures on boundaries that are integer aligned.

```
int      i;
short int iarray[6];
char     carray[12];
struct msg {
         char        m1;
         char        m2;
         short int   m3;
         long        m4;
         } msga;
```

The following calls will send the specified message to the mailbox of the task with id *taskid* or to the message exchange with exchange id *mxid* . The messages are sent at priority 3.

```
int priority = 3;
ajsend(taskid, priority, &c);          /* Send c                  */
ajsend(taskid, priority, &i);          /* Send i                  */
ajsend(taskid, priority, iarray);      /* Send iarray             */
ajmxsnd(mxid, priority, carray);       /* Send carray             */
ajmxsnd(mxid, priority, &msga);        /* Send msga               */
```

The messages are stored in FIFO order in the task mailbox or message exchange message queue. The caller specifies the priority level of the message. The priority level (0, 1, 2 or 3) determines the message queue (mailbox) into which the message will be placed.

If the caller is a task sending a message to another task, it can request AMX to suspend its operation until the called task has executed in response to the request. This is accomplished by calling *ajsenw* (or its variation *ajsenwp*) rather than *ajsend*.

AMX uses message envelopes for parameter transmission. AMX gets a free envelope, moves the caller's message parameters into it and adds the envelope to the task mailbox or message exchange message queue.

If the sender requested to wait, AMX inserts information into the envelope which allows AMX to remember that the sending task has been suspended waiting for acknowledgement of the receipt of its message by some other task.

Note that when AMX returns from a call to send a message, the memory occupied by the caller's message is available for reuse by the caller.

> Note
>
> AMX always copies *MAXMSZ* bytes from the sender's message into the envelope. *MAXMSZ* is defined in your AMX System Configuration Module and is >= 12.
>
> Hence, the transmitted message will always be *MAXMSZ* bytes in length even if your message is a single integer.

For assembly language programmers, procedure *AASEND* is used to send a message to a task and to wait if desired.

```
USER_CODE SEGMENT BYTE 'CODE'
;
          ASSUME    CS:USER_CODE
;
MSGA      DB        ?                ;m1
          DB        ?                ;m2
          DW        ?                ;m3
          DD        ?                ;m4
;
MSGAD     DD        MSGA             ;pointer to MSGA
TASKID    DW        ?                ;taskid
;
USERPROC  PROC      FAR
          LES       BX,CS:MSGAD      ;ES:BX = A(MSGA)
          MOV       DX,CS:TASKID     ;DX = task id
          MOV       CH,0             ;CH = 0 (no wait)
;                                    ;CH = 80H (wait)
          MOV       CL,2             ;CL = priority (0,1,2 or 3)
          CALL      AASEND           ;send the message
          RET
;
USERPROC  ENDP
;
USER_CODE ENDS
```

Figure 3.9-1 provides an example of the manner in which messages are allowed to queue on a message exchange. The same message queuing technique is used when messages are sent to a task's private mailboxes within its Task Control Block (TCB).

In the example, the following situation is assumed to exist at the message exchange on which the destination task is waiting.

The message exchange has four message queues corresponding to the four priority levels at which it can post messages. Level 0 is the highest priority; level 3 is the lowest priority. The example shows that no messages are pending on levels 0 or 2. Three messages, I, J and K, are pending on level 1. Two messages, L and M, are pending on level 3. One message, X, is currently being serviced by the destination task.

The destination task has been interrupted and it is ready to resume execution. As a result of the interrupt, task YY was allowed to execute because it was of higher priority than the destination task. Task YY made a request to AMX to send a message to the message exchange at message priority level 1. As a result of this request, AMX moved the message parameters from task YY into a message envelope and added this envelope to the bottom of the level 1 message queue associated with the message exchange. The result is shown in Figure 3.9-1.

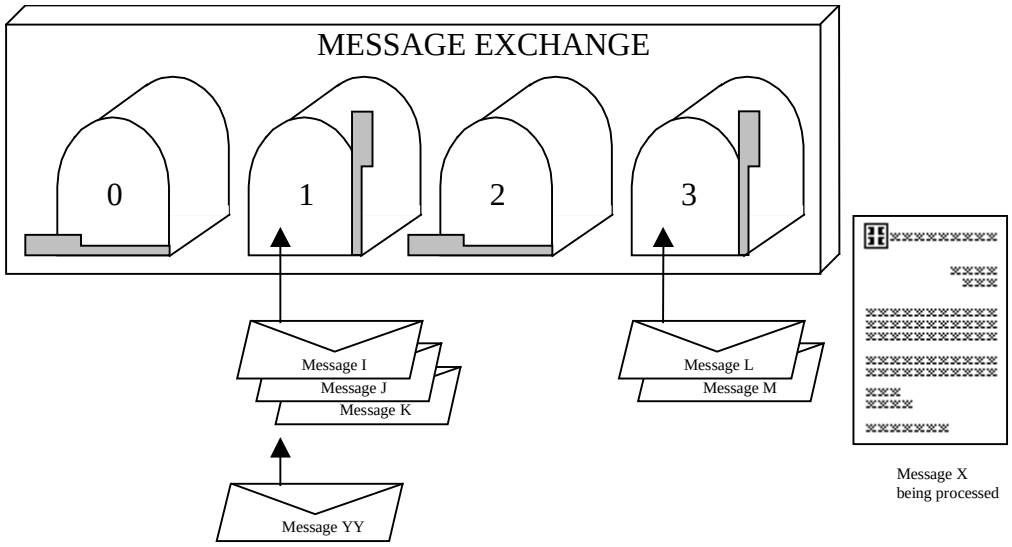


Figure 3.9-1 Message Transmission

In due course the destination task will continue to execute and complete its processing of the current message, X. The destination task will then request AMX for a message from the message exchange three more times in succession to process messages I, J and K. Finally, the message from task YY will be retrieved from the message exchange and processed.

Figure 3.9-2 illustrates the situation at the destination task at the instant it begins to process the message from task YY. Provided that no additional calls to post messages to the message exchange have occurred, the messages will be as shown. Message queues 0, 1 and 2 are empty. Messages L and M are still pending at priority level 3. The message from task YY is ready for processing by the destination task.

When the destination task retrieves a message from the message exchange, the message parameters are already removed from the envelope and copied to the storage area provided by the task, usually on the task's stack. The message parameters received by the destination task are in exactly the same order as they were sent by the caller.

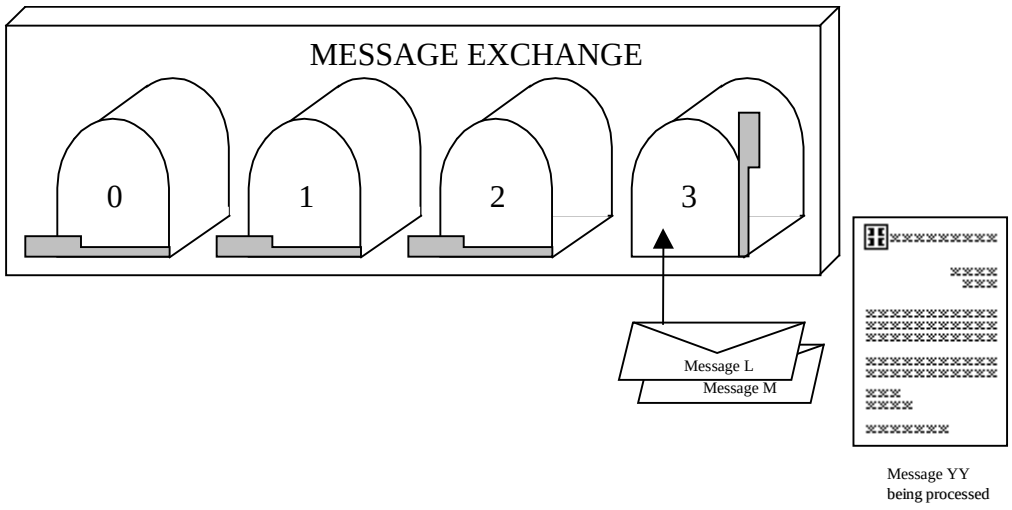


Figure 3.9-2 Message Reception

It is important to note that a copy of the sender's message parameters is sent to the destination task. Once the sender's parameters have been copied into the message envelope, the caller is free to reuse the parameter storage if desired. Thus, as soon as procedure *ajsend* or *ajmxsnd* (or any of their variations) returns to its caller, the parameter variables are free for reuse.

A message which is delivered to a task in one of its mailboxes is copied directly to the task's stack prior to starting the task. However, when a message exchange is used, the destination task must provide a pointer to storage large enough to hold the entire message which it will receive. Since AMX unconditionally delivers messages of *MAXMSZ* (>= 12) bytes in length, the message storage must be of at least that size.

The AMX task which receives a variety of messages should declare a union *rmsg* in order to reference the different messages. Note that the union should include one instance of an array *amxmsg* consisting of three *long* values to ensure that *rmsg* is large enough to hold any AMX message.

```
union rmsg {
         long       amxmsg[3];
         char       c;
         int        i;
         short int  iarray[6];
         char       carray[12];
         struct msg msga;
         };
```

Then the task can be written as follows:

```
void taskn(int dummy)
{
   union rmsg *pmsg;

   pmsg = (union rmsg *)&dummy;     /* Pointer to received message    */
   :
   :
   The received message can now be accessed
   using union pointer pmsg.
   :
   }
```

In this simple example, the task receiving the messages has no obvious way of determining how to interpret the message. Is the message one character (*pmsg->c*) or a whole structure (*pmsg->msga*)? This dilemma is usually solved by including an application specific operation code with all messages.

Note that we have declared the task's received message as an integer and then used pointer variable *pmsg* to access the parameters in the message. AMX passes parameters by value with no concern for the restrictions imposed by various high level languages. Hence a whole array or structure can be passed by value by AMX to a task. However, not all C compilers support such parameter passing.

Message Extension

AMX delivers messages to a task by copying the message from a message envelope to a message frame on the task's stack. AMX also presents an extension to the application message on the task's stack. The message extension identifies who sent the message and, if a task, whether the sender is waiting for a response.

The message extension is provided for the use of private AMX tasks such as the Kernel Task. Application tasks should rarely be aware of its presence.

Access to the message extension is illustrated in the following C example. Refer to Appendix D for a description of the message extension structure *amxmsgxs*. Assembler programmers will find the message extension on the task stack at *SS:BP+UMS* where *UMS* is the size of your application messages declared in your AMX User Parameter File.

```
#include "amx831sd.h"              */ AMX Structure Definitions     */
#define UMS 12                     */ User message size             */

void taskn(int dummy)              */ Dummy parameter declaration   */
{
   struct amxmsgxs *msgxp;         /* Message extension pointer     */

   msgxp = (struct amxmsgs *)((char *)&dummy + UMS);
   :
   The message extension can now be accessed
   using pointer msgxp.
   :
   }
```

3.10 Restart Procedures

The manner in which the operating system begins execution is application dependent. Execution begins in the user domain providing the opportunity for hardware specific and application dependent setup prior to the initialization of the AMX system. For example, hardware interfaces may require custom configuring. In some systems, it might be desirable to perform a memory integrity check before system startup is permitted.

AMX will enable or disable the interrupt system at the time you launch AMX, as dictated by the launch parameter you provide when you start AMX.

Once AMX has initialized all of its internal variables and structures, it executes the sequence of application Restart Procedures provided in the Restart Procedure List in your System Configuration Module (see Chapter 14.6). These procedures can invoke AMX services to start tasks and initialize interval timers.

When AMX calls a Restart Procedure, the following conditions exist:

> Interrupts reflect the state specified at the time AMX was launched. If AMX was launched with launch parameter *AMLPIE*, interrupts will be enabled. Otherwise interrupts will be disabled.
>
> All registers are free for use.
> *DS,ES* *DGROUP* segment
> *SS:SP* AMX Kernel Stack ready for use
> The direction flag is set to forward.

Restart Procedures are written as Large or Medium model C procedures without formal parameters.

```
void cdecl rrproc(void)                    /* Restart Procedure        */
{
   :
   Do restart processing
   :
   }
```

A Restart Procedure is coded in assembly language as a *FAR* procedure as follows:

```
USER_CODE SEGMENT BYTE 'CODE'
;
            ASSUME   CS:USER_CODE
;
RRPROC      PROC     FAR
            :
            Restart Procedure code goes here.
            :
            RET
;
RRPROC      ENDP
;
USER_CODE   ENDS
```

Restart Procedures can enable specific device interrupts if required. Note that interrupts from a device should not be enabled until the application ISP has been installed and made ready to handle the interrupting device.

Restart Procedures should not enable the processor interrupt system if your launch parameter indicated that AMX was to execute with interrupts disabled during the launch. In this case, AMX will disable interrupts upon completion of your Restart Procedure.

A Restart Procedure must not issue any AMX directives which would in any way suspend or terminate task execution.

Restart Procedures use the AMX Kernel Stack. In addition to the minimum stack size required for the AMX Kernel Stack, you must allocate sufficient stack to satisfy the worst case requirements of all application Restart Procedures.

> Note
>
> Restart Procedures must only use AMX services which are marked in Chapter 16 as
> ■ Restart Procedure

3.11 Exit Procedures

An AMX system can be shut down in an orderly fashion by a task call to procedure *ajexit*. The manner in which the operating system ends execution is application dependent. For example, hardware interfaces may require restoration to their initial states.

AMX supervises the shutdown process by sequentially calling all of the application Exit Procedures in the order defined in the Exit Procedure List in your System Configuration Module (see Chapter 14.6). Once all Exit Procedures have been executed, AMX returns to the original program that launched AMX with an *ajentr* call.

When AMX calls an Exit Procedure, the following conditions exist:

Interrupts are enabled.
All registers are free for use.
DS,ES *DGROUP* segment
SS:SP Task stack which was in effect when *ajexit* was called.
The direction flag is set to forward.

Exit Procedures are written as Large or Medium model C procedures without formal parameters.

```
void cdecl epproc(void)                    /* Exit Procedure          */
{
   :
   Do exit processing
   :
   }
```

An Exit Procedure is coded in assembler as a *FAR* procedure as follows:

```
USER_CODE SEGMENT BYTE 'CODE'
;
            ASSUME   CS:USER_CODE
;
EPPROC      PROC     FAR
            :
            Exit Procedure code goes here.
            :
            RET
;
EPPROC      ENDP
;
USER_CODE ENDS
```

Exit Procedures execute in the context of the task which issued the *ajexit* call. They are therefore free to use all services normally available to tasks. For instance, an Exit Procedure could use *ajsenw* to send a shutdown message to a task and wait for that task to do its shutdown processing. When the Exit Procedure resumes after the *ajsenw* call, it returns to AMX which then calls the next Exit Procedure in the list.

An Exit Procedure must not issue any AMX directives which would in any way terminate task execution.

Exit Procedures use the stack of the task which called *ajexit*. It is therefore essential that the task which calls *ajexit* be allocated sufficient stack to satisfy the worst case requirements of all application Exit Procedures.

> Note
>
> Exit Procedures must only use AMX services which are marked in Chapter 16 as
> ■ Exit Procedure

> Warning
>
> If any Exit Procedure is coded using the Medium model, the task which calls *ajexit* MUST also use the Medium model.

3.12 Task Enhancements

AMX offers several task enhancements which, although rarely used, can occasionally come in handy. These enhancements are briefly summarized below.

Task Control Block Extension

Within a task's Task Control Block, 16 bytes are reserved for the private use of the task. Their use is completely determined by the application.

A Task Control Block for a particular task can be located using procedure *ajtktcb*. The private storage is located at offset *AMTCBUSR* in the Task Control Block.

If you are coding in C, you must use structure definition *amxtcbs* from header file *AMX831SD.H* to access this storage.

If you are coding in assembly language, you must use structure definition *AMXTCBS* from header file *AMX831SD.DEF* to access this storage.

Stack Fences

When AMX creates a task, it installs a fence at the top and bottom of the task's stack. The fence is a 32-bit value containing the unusual pattern *0x55555555* ('*UUUU*').

The 4-character task tag is copied to the stack immediately below the top fence as a debugging aid to identify each stack's owner.

These fences can be of significant help during the development and testing of your AMX system. When you are trying to detect a fault in your system, it is often advisable to examine your task stacks carefully to be sure that none of the fences have been altered. An altered fence is an indication that you have a task which is overflowing or underflowing its stack or, worse yet, a bad piece of code which is writing to some task's stack in error.

AMX Kernel Task Priority

The AMX Kernel Task operates as the highest priority task in your AMX system. Its execution priority is zero. It has the reserved task tag `'AMXK'`.

Occasionally, an application is encountered in which a task must execute at very high priority in order to cope with the idiosyncrasies of an extremely high speed device. In such cases, the priority of the AMX Kernel Task can be dropped from zero to some lower priority determined by you for your application. To do this, you must call AMX procedure `ajtkpry` to change the priority of the AMX Kernel Task.

It is recommended that you exhaust all other avenues of approach to the high speed device handling problem before taking this way out. Usually, high speed devices are best handled at the interrupt level using the techniques described for special interrupts in Chapter 4.7.

If you choose to drop the priority of the AMX Kernel Task, you must pay a penalty. Any task which you place at a priority higher than the AMX Kernel Task is not allowed to use any of the following AMX services:

Raise privilege level
Timed waits
Interval timer operations
Semaphore Manager services
Event Manager services
Message Exchange Manager services
Memory Manager services

This page left blank intentionally.

4. Interrupt Service Procedures

4.1 The Processor Interrupt Facility

The key to event-driven, real-time, multitasking systems is the processor's interrupt facility. Tasks execute with the interrupt facility enabled permitting the system to respond to a real-time event.

The hardware interrupt mechanism is an automatic facility provided by the processor. AMX permits the system designer to determine how the hardware interrupt facility will be employed.

Tasks must execute with the interrupt facility enabled.

From time to time, AMX must inhibit interrupts while it performs a critical, indivisible sequence of operations. AMX keeps such intervals very short. For instance, even while AMX is switching from one task to another, it is able to respond to interrupts.

To further improve interrupt response, AMX permits nesting of interrupts on processors which support this feature. As soon as the interrupt request has been cleared, interrupts can be enabled to permit response to other external events.

When an interrupt occurs, the processor disables the interrupt facility and stores the flags and return address on the current stack to permit the processor to eventually resume the interrupted process. In some cases, an internal or external interrupt mask is set to inhibit new requests of priority less than or equal to the priority of the device being serviced.

Hardware external to the processor usually identifies the interrupt source. The processor uses this identification to automatically fetch a unique device dependent pointer from the processor vector table and branch to the address specified by that pointer. The program located at that address is called an Interrupt Service Procedure (ISP).

In general, an ISP saves the registers it wishes to use, services the device, restores the registers, enables the interrupt system and returns to the executing program at the point of interruption.

If more than one device is connected to the same external interrupt source, your ISP must determine the interrupt source in one of several ways. The simplest, but slowest, is a software poll of the devices. The ISP tests each device sequentially to determine the source of the interrupt and branches to a device service procedure to handle the specific interrupt.

Alternatively, external hardware can be added to provide unique vectoring for each device which can generate interrupts. If this approach is adopted, then a separate ISP must be provided for each device.

The AMX Interrupt Supervisor simplifies ISP operation within the AMX multitasking environment. The AMX Interrupt Supervisor permits an ISP to communicate with any task in the system. The remainder of this chapter describes how ISPs are used within a system.

When used with the 8086 family of microprocessors, AMX supports the following exception and interrupt sources:

Divide error	(*DIV* or *IDIV* instruction)
Non-Maskable interrupt	(*NMI* hardware signal)
Overflow	(*INTO* instruction)
Bounds error	(*BOUND* instruction)
External devices	(*INTR* hardware signal)

All other special processor exceptions such as invalid opcode (type 6) or the 80386 general protection exception (type 13) are not handled by AMX. They must be specifically serviced by your own exception handler.

The design and implementation of the interrupt structure used in a particular application is completely under your control. Significantly, AMX does not use any of the software interrupts such as the Single-Step or INT n interrupts. They are left for application and testing use.

4.2 ISPs for External Interrupts

Two types of ISPs exist: nonconforming ISPs and conforming ISPs.

Nonconforming ISPs

A nonconforming ISP must quickly service the device to remove the interrupting condition. The ISP must preserve all registers which it uses. The nonconforming ISP cannot make calls to any AMX service procedures.

The nonconforming ISP executes in the context of the process (task, ISP, AMX kernel) which was interrupted. The ISP therefore uses the stack of the interrupted process. Consequently, all stacks must be large enough to meet the worst case needs of all nonconforming ISPs.

The nonconforming ISP can enable interrupts to allow interrupt service by other higher priority nonconforming ISPs. The nonconforming ISP **MUST NOT** allow an interrupt to a conforming ISP.

Conforming ISPs

A conforming ISP is intended for use with AMX. A conforming ISP consists of an ISP root and an Interrupt Handler. Often these two components are merged indistinguishably into one body of code. The processor vectors to the ISP root which informs the AMX Interrupt Supervisor that the interrupt has occurred. The Interrupt Supervisor preserves the state of the interrupted task and, if necessary, switches to an interrupt stack. Upon return to the ISP root, the associated Interrupt Handler is called. Your handler must perform all services required by the device. Your handler is free to make procedure calls to a subset of the AMX service facilities.

The conditions that will exist upon entry to your Interrupt Handler are dependent upon the tool set which you are using and the language in which your Interrupt Handler is programmed. In general:

Interrupts are disabled (masked) at a device dependent priority level.
A subset of the registers are free for use.
The AMX Interrupt Stack is in use.

When your Interrupt Handler executes, external interrupts of priority less than or equal to the interrupting device are always disabled. Your Interrupt Handler must NOT enable any interrupts of lesser or equal priority. Your handler can enable higher priority external interrupts if permitted by your target hardware and your application software design.

When device service is completed, your Interrupt Handler returns to the ISP root which informs the AMX Interrupt Supervisor that interrupt service is complete.

AMX monitors calls made by the Interrupt Handler to AMX service procedures. If no such calls have been made, the AMX Interrupt Supervisor automatically restores the state of the interrupted task and allows the ISP root to return directly to the interrupted task at its point of interruption.

If the Interrupt Handler requested AMX to initiate or resume execution of some task of higher priority than the interrupted task, the AMX Interrupt Supervisor suspends the interrupted task and marks it as ready to resume execution at the earliest opportunity. The AMX Task Scheduler is then invoked to determine the highest priority task capable of execution.

If interrupts nest, the Interrupt Supervisor defers its task switching checks until all of the concurrent interrupts have been serviced.

In due course, AMX returns to the ISP root ready to resume execution at the point of interruption. It is **important** to note that, because of task switching invoked by the AMX Interrupt Supervisor, there may be a significant delay before AMX returns to the ISP root and resumes execution of the interrupted task.

> Note
>
> Interrupt Handlers must only use AMX services which are marked in Chapter 16 as
> ■ ISP

The Interrupt Handler is free to use the following AMX service procedures to communicate with tasks. If the affected task is of higher priority than the task which was preempted by the interrupt, AMX will initiate a task switch when the interrupt service has been completed.

ajsend	*AASEND*	Request a task to execute and send it a message
ajtrig	*AATRIG*	Request a task to execute with no message
ajwake	*AAWAKE*	Wake a task known to be waiting for this interrupt
ajsgnl	*AASGNL*	Signal a task
ajsmsig	*AASMSIG*	Signal to a semaphore
ajevsig	*AAEVSIG*	Signal an event
ajmxsnd	*AAMXSND*	Send a message to a message exchange

Interrupt Handlers are also free to use the following AMX buffer management and timing facilities.

ajbget	*AABGET*	Get a buffer
ajbfre	*AABFRE*	Free a buffer
ajbau	*AABAU*	Add to buffer use count
ajtmwr	*AATMWR*	Start/stop an interval timer
ajtmrd	*AATMRD*	Read an interval timer

The full range of AMX circular list handling routines (see Chapter 11) can be used by application Interrupt Handlers. These circular lists can be especially useful to provide character buffering. For example, an input device Interrupt Handler can add characters to the bottom of a circular list while a related task removes them from the top of the list.

The AMX List Manager services (see Chapter 12) can also be used by Interrupt Handlers. Note that an Interrupt Handler should not manipulate keyed lists.

Conforming ISP Construction

The construction of an Interrupt Service Procedure (ISP) to service an external interrupt is a simple process.

The conforming ISP consists of two parts: the ISP root and your Interrupt Handler. The **ISP root** is a code fragment most easily created using procedure *ajispm*.

The **Interrupt Handler** can be written in as a Large or Medium model C function without formal parameters. If the C function makes any calls to AMX, restrictions for ISPs must be obeyed. The following example of a device Interrupt Handler illustrates how little application code must be programmed to satisfy AMX.

If the processor Interrupt Vector Table (IVT) is in RAM and AMX was launched with vector access allowed, you can install the pointer to the ISP root into the IVT using AMX function *ajivtw*. Otherwise, you must initialize the device's entry in the IVT as described in Chapter 4.8 before launching AMX. The following example assumes that the device interrupts through vector number 32.

```
#include "amx831cf.h"                    /* AMX C Interface Header   */
static struct amxisps isproot;           /* ISP root                 */

void cdecl dvcisp(void)
{
   local variables, if required
   :
   Perform all device service.
   Interrupts may be enabled using ajei() if nested interrupts
   are supported by your design.
   :

   /* If necessary, disable interrupts and clear the request in      */
   /* one or more interrupt controllers (such as the Intel 8259).    */

   ajdi();                               /* Disable interrupts       */
   ajoutb(0x20, 0x20);                   /* Non-specific EOI         */
   }


void cdecl rrproc(void)                  /* Restart Procedure        */
{
   ajispm(dvcisp, &isproot);             /* Make ISP root            */

   ajivtw(32, (void (*)())&isproot);     /* Install ISP in IVT       */
   }
```

The ISP root created by AMX in static variable *isproot* is an ISP coded as follows:

```
ISPROOT  LABEL    FAR
         CALL     AAINT            ;begin interrupt service
         CALL     DVCISP           ;ISP
         CALL     AAINX            ;end interrupt service
         IRET                      ;end of interrupt
```

The Interrupt Service Procedure should be coded using assembly language if speed of execution is critical.

```
            EXTRN     AAINT:FAR
            EXTRN     AAINX:FAR
;
DVC_CODE SEGMENT BYTE 'CODE'
;
            PUBLIC    DVCISP
; The ISP is located in user program memory
;
            ASSUME    CS:DVC_CODE
;
DVCISP      PROC      FAR
            CALL      AAINT               ;let AMX know
            :
            :
            Perform all device service.
            All registers are free for use.
            Interrupts may be enabled if nesting is supported.
            :
            :
            If necessary, disable interrupts and clear the request in
            one or more interrupt controllers (such as the Intel 8259).
            CLI                           ; Disable interrupts
            MOV       AL,20H
            OUT       20H,AL              ; Non-specific EOI
            CALL      AAINX               ;let AMX know
            IRET                          ;return from interrupt
;
DVCISP      ENDP
;
DVC_CODE    ENDS
```

If the processor Interrupt Vector Table (IVT) is in RAM and AMX was launched with vector access allowed, you can install the pointer to the ISP into the IVT using AMX function *AAIVTW*. The ISP begins execution at address *DVCISP*. The ISP calls the AMX Interrupt Supervisor (*AAINT*) to inform AMX that the interrupt has occurred.

The AMX Interrupt Supervisor, when invoked, switches to an Interrupt Stack provided by you in your System Configuration Module. The AMX Interrupt Supervisor then returns to the application ISP. The ISP must perform all services required by the device.

Upon return to the ISP from the AMX Interrupt Supervisor entry procedure *AAINT*, the registers are initialized as follows.

Interrupts are disabled.
All registers are free for use.

DS,ES	*DGROUP* Segment
SS:SP	AMX Interrupt Stack ready for use
CX,DX,SI	Unaltered by procedure *AAINT*

The direction flag is set to forward.
All other registers are undefined.

It should be noted that AMX procedure *AAINT* preserves registers *CX*, *DX* and *SI*. This feature can be used to advantage when creating special ISPs like those described in Chapter 4.7.

The external interrupt facility is disabled. Your ISP can enable the external interrupt facility if so desired. In this case, the ISP must be sure to either clear the source of the interrupt before enabling interrupts or to mask (disable) the interrupt source if the hardware exists to do so.

When the interrupt service is complete, the ISP informs the AMX Interrupt Supervisor by issuing the termination call to *AAINX*. The final *IRET* instruction restores the processor flags register and returns to the point of interruption.

It is **important** to note that, because of task switching invoked by *AAINX*, there may be a significant delay before AMX returns to your *IRET* to resume execution of the interrupted task. No device operations, AMX calls or commands to an interrupt controller such as the Intel 8259 chip are allowed after the call to *AAINX*.

4.3 Nested Interrupts

AMX supports nested interrupts. The AMX Interrupt Supervisor maintains a private nesting level indicator. AMX must be informed of the start (*AAINT*) and end (*AAINX*) of each interrupt.

When AMX sees that a task has been interrupted, it switches to a predefined Interrupt Stack. If AMX detects that the interrupt has occurred during execution of a device Interrupt Service Procedure, then no stack switching occurs. The interrupted ISP is suspended and the new ISP is started.

When an ISP ends, AMX takes action based on the state of its nesting indicator. When a nested interrupt ISP is completed, AMX returns to the interrupted ISP. When the last interrupt ISP (corresponding to the first task interrupt) is completed, AMX prepares to return to the interrupted task. If, as a consequence of interrupt service, a significant event has been declared, AMX suspends the interrupted task and goes to the AMX Task Scheduler to find the highest priority task which is ready to execute.

Since the AMX Interrupt Stack is used for nesting interrupts, the stack size must be large enough to support the worst case combination of nested ISPs. Each nested interrupt requires that the Interrupt Stack be increased to meet the needs of the additional nested ISP.

> Warning
>
> You must not permit a conforming ISP to interrupt a nonconforming ISP.
>
> The AMX clock ISP is a conforming ISP.

4.4 ISP/Task Communication

AMX provides a set of service procedures to ease the communication between tasks and device Interrupt Handlers. These AMX procedures simplify event synchronization and permit parameter passing to tasks.

Wait/Wake Synchronization

The *ajwait*/*ajwake* pair of procedures is often used for event synchronization. A task issues an *ajwait* call to AMX to wait unconditionally for an event. When the event occurs, as indicated by an interrupt, the Interrupt Handler makes an *ajwake* call identifying the task which it wishes to wake up.

Synchronization capability can be further enhanced using the automatic timeout facility provided by AMX. The task issues an *ajwatm* call indicating the maximum interval for which it is willing to wait for the event. When the interrupt occurs, the Interrupt Handler issues the *ajwake* call to wake up the task. The task resumes execution with an indication provided to it by AMX that the event did occur. If, on the other hand, the interrupt does not occur within the specified timeout interval, the AMX Kernel Task will wake up the task. In this case, the task resumes execution with an indication provided by AMX that a timeout occurred. When the task resumes execution, it is, therefore, capable of determining if the event occurred within the expected interval.

Task Signal Synchronization

The simple wait/wake synchronization described above is actually a subset of event synchronization provided by AMX known as task signals. Each task has associated with it a set of 15 independent flags called task signals. The task can wait for any specific task signal or combination of task signals by calling *ajsgwat* indicating the maximum interval which it is willing to wait.

When the interrupt associated with a particular task signal occurs, the Interrupt Handler issues the *ajsgnl* call to signal the task that the particular event has occurred. If the task is not still waiting for other task signals, AMX will allow the task to resume execution with an indication that its task signal conditions were met. If, on the other hand, all task signals for which the task was waiting do not occur within the specified timeout interval, the AMX Kernel Task will force the task to resume execution with a timeout indication.

Semaphore Synchronization

The AMX Semaphore Manager provides an even more powerful synchronization capability. It provides the automatic timeout facility and also allows more than one task to wait for the same event, with each task determining its own waiting priority. Furthermore, the Interrupt Handler need not know the identity of the waiting tasks (if any) or their chosen waiting priorities. The AMX Semaphore Manager will automatically ensure that the task with the highest waiting priority is given access to the semaphore first.

Synchronization using counting semaphores with an initial count of zero is achieved as follows. A task issues an *ajsmwat* call indicating the maximum interval for which it is willing to wait for the semaphore and its chosen waiting priority. When the interrupt occurs, the Interrupt Handler issues an *ajsmsig* call to wake up the task with the highest waiting priority that is blocked on the semaphore. The task will always know when it resumes execution whether the event actually occurred or whether the maximum wait interval elapsed.

Event Group Synchronization

The AMX Event Manager provides another form of synchronization. It allows more than one task to be waiting for a specific event or for a combination of events. It also provides the automatic timeout facility. The Interrupt Handler does not have to know which tasks (if any) are waiting for its event.

Synchronization using an event group is achieved as follows. A task clears an event flag in an event group and then issues an *ajevwat* call indicating the maximum interval it is willing to wait for the event to occur. The Interrupt Handler issues an *ajevsig* call to set the particular event flag for which it is responsible. The Event Manager then allows all tasks which are waiting for that particular event flag to be set to resume execution. The task will always know when it resumes execution whether the event actually occurred or whether the maximum wait interval elapsed.

Message Exchange Synchronization

An Interrupt Handler can communicate with a task by sending a message to a message exchange. The AMX Message Exchange Manager allows more than one task to wait for a message from a message exchange with each task determining its own wait priority and maximum wait interval. An Interrupt Handler does not need to know the identity of the waiting tasks (if any) or their chosen waiting priorities.

Synchronization via a message exchange is achieved as follows. A task calls *ajmxwat* to wait on a particular message exchange indicating the priority of its request and the maximum interval it is willing to wait for a message to arrive.

When the interrupt occurs, the Interrupt Handler calls *ajmxsnd* to send a message to the message exchange. The Message Exchange Manager delivers the message to the task and allows the task to resume execution. If no interrupt occurs before the timeout interval expires, the AMX Kernel Task will force the task to resume execution with a timeout indication.

Task Triggering

An Interrupt Handler can communicate with a task by invoking the task's execution. When an interrupt occurs, the Interrupt Handler issues the *ajtrig* call to AMX identifying the task which it wishes to have executed.

This technique can be very useful for handling slowly occurring events. For example, a device generates an interrupt and the Interrupt Handler responds by acquiring a block of data from the device. The data is stored in memory by the Interrupt Handler for subsequent use by a task. The Interrupt Handler then starts the task with the *ajtrig* call. It is assumed that timing is such that the task will be able to completely process the data prior to the next occurrence of a similar event. This constraint is typical in many real-time system implementations.

Message Transmission

An Interrupt Handler can also start a task and send it a message. For example, a control panel might be used by an operator to initiate actions within a system. An interrupt is generated when the operator depresses a pushbutton requesting a specific function. In response to the interrupt, the Interrupt Handler reads a command register at the control panel to determine the action to be taken. Data, such as set-point settings, would also be read by the Interrupt Handler.

The Interrupt Handler interprets the command, determines the task in the system responsible for performing the requested function, and issues an *ajsend* call to AMX requesting execution of that task. The data retrieved from the control panel is transmitted to that task as parameters in the message.

Chapter 3.9 provides a complete description of the parameter passing process.

4.5 Task Error Traps

The 80x86 processor automatically detects the occurrence of execution errors such as division by zero, arithmetic overflow or an array bound violation. These errors, by their very nature, must be handled by the application in the context of the task in which they occur.

AMX offers tasks a convenient method of trapping such errors. For each task, AMX maintains a pointer to a task trap handler for each detectable error.

Normally, when a task is running, AMX treats these errors as fatal if they occur. A task can specify a trap handler for a particular error trap by calling AMX service procedure *ajitrp* or *AAITRP*. Thereafter, if the error occurs while running in the context of the task, the task automatically branches to its trap handler.

Note that the actual processor error exception is completely serviced by AMX. The task simply appears to have suddenly jumped to its trap handler.

Divide errors cannot be disabled. Once a task specifies a divide error trap handler, any subsequent divide error which occurs when that task is executing will cause an immediate jump to the task's divide error trap handler.

Overflow error exceptions do not occur automatically. The task must execute an interrupt on overflow (*INTO*) instruction to cause the overflow exception if the overflow flag (*OF*) is set. If the exception occurs and the task has specified an overflow error trap handler, then AMX will cause the task to jump to that handler.

Bound error exceptions do not occur automatically. The task must execute a bound check (*BOUND*) instruction to cause the bound error exception if specific array bounds are violated. If the exception occurs and the task has specified a bound error trap handler, then AMX will cause the task to jump to that handler.

AMX maintains unique divide, overflow and bound error trap handlers for each task. If a task is suspended and another task executes, AMX selects the divide, overflow and bound trap handlers for the new task. When the suspended task resumes, its unique trap handlers are reinstated.

A task trap handler for a particular error trap is reset (cancelled) with a request to AMX to set the pointer to *NULL* (*0L*).

When a task is first created, AMX sets the task's trap handlers to *NULL* forcing errors to be fatal until the task specifies its own trap handlers. Once a task defines its trap handlers, they remain defined even if the task ends execution. If the task is subsequently executed again, its previously defined trap handlers remain in effect unless altered by the task.

If a trapped error exception occurs in a task with no trap handler or in a Restart Procedure, an ISP or a Timer Procedure, AMX initiates a fatal exit as described in Chapter 13.1.

The task trap handler can be written as a Large or Medium model C function with formal parameters.

```
#include "amx831sd.h"                    /* AMX Structure Definitions*/

void cdecl tdiverr(
struct amxregs regs,                     /* Register structure       */
void FAR *faultp)                        /* Fault pointer            */
{
   :
   Process the error
   :
   }
```

Since the task trap handler executes in the context of the task, the task's stack must account for the stack used by the handler. An additional *sizeof(struct amxregs)* bytes of stack is required to accommodate the processor dependent stack frame generated by AMX prior to its call to the trap handler.

AMX provides the address of the fault and the state of each processor register at the time of the fault.

The error exception is serviced by AMX. The address of the fault is on the stack as parameter *faultp*. For the overflow trap, the address points to the instruction following the *INTO* instruction. For the 8086/88 and 80186/188 *DIV*, *IDIV* or *BOUND* instruction, the address points to the instruction following the instruction which caused the fault. For the 80286 and 80386 *DIV, IDIV* or *BOUND* instruction, the address points to the instruction (including any instruction prefixes) which caused the fault. The state of each register at the time of the fault is present on the stack in an AMX register structure *regs* (see definition in Appendix D.1).

The register values in structure *regs* can be examined and modified with care. If necessary, the *FAR* pointer *faultp* can be modified, with care, to force resumption at some other location in the task code. If the trap handler returns to AMX, execution will resume at the location specified by *faultp* with registers set according to the values in structure *regs*.

Since the trap handler executes in the context of the task in which the error trap occurred, it is free to use all AMX services normally available to tasks. In particular, the trap handler can call *ajend* to end task execution if so desired.

> Note
>
> Task trap handlers are NOT Interrupt Handlers even though the error interrupt is detected via a processor exception.

Note that the C trap handler receives a structure *amxregs* passed by value. If your C compiler does not allow such declarations, you can use casts to coerce access to the parameters on the stack as follows:

```
#include "amx831sd.h"                    /* AMX Structure Definitions*/

void cdecl tdiverr(int dummy)
{
   struct amxregs *regp;                 /* Pointer to registers     */
   void FAR *faultp;                     /* Fault pointer            */

   regp = (struct amxregs *)&dummy;
   faultp = *((void FAR **)(regp + 1));
   :
   Process the error
   :
   }
```

The task trap handler can be coded in assembly language as a *FAR* procedure. Upon entry to the procedure, the following conditions exist:

Interrupts are as they were at the time of the fault.
All registers are free for use.

DS,ES	*DGROUP* Segment
SS:SP	Task stack ready for use
SS:BP	A(parameters on stack)

The direction flag is set to forward.

The parameters on the stack include a register structure *AMXREGS* (see Appendix D.2) followed by a *FAR* pointer to the fault location. For the overflow trap, the address points to the instruction following the *INTO* instruction. For the 8086/88 and 80186/188 *DIV*, *IDIV* or *BOUND* instruction, the address points to the instruction following the instruction which caused the fault. For the 80286 and 80386 *DIV*, *IDIV* or *BOUND* instruction, the address points to the instruction (including any instruction prefixes) which caused the fault.

If the overflow trap handler issues a *FAR* return instruction, it will cause execution to resume following the *INTO* instruction. If the divide or bound trap handler issues a *FAR* return instruction, the effect will be processor dependent. The 8086/88 and 80186/188 will resume execution following the instruction which caused the fault. For the 80286 and 80386, the instruction which caused the fault will be executed again. Note that recovery from divide and bound errors may be next to impossible in some applications.

```
            INCLUDE AMX831SD.DEF          ;AMX Structure Definitions
;
USER_CODE SEGMENT BYTE 'CODE'
;
            ASSUME      CS:USER_CODE
;
TDIVERR     PROC        FAR
            :
            Access register structure AMXREGS via SS:BP
            :
            LES         BX,DWORD PTR [BP+(SIZE AMXREGS)]
;                                         ;ES:BX = A(fault)
            :
            Process the error
            :
            RET
;
TDIVERR     ENDP
;
USER_CODE   ENDS
```

4.6 Non-Maskable Interrupt

The Intel 80x86 processor and equivalents provide a non-maskable interrupt (NMI). This interrupt cannot be inhibited by software.

You have complete control over the non-maskable interrupt ISP. Usually, the NMI interrupt is used to signal a catastrophic event such as a pending loss of power. The ISP must process the interrupt in an application-dependent fashion, restore all registers and return to the point of interruption if feasible. This ISP must assure that the interrupt facility is restored according to its state at the time the non-maskable interrupt occurred.

The NMI ISP must be a nonconforming ISP. The NMI ISP cannot use AMX services. Consequently the non-maskable interrupt cannot be used as an additional, general purpose device interrupt.

Some hardware assisted debuggers may use the NMI interrupt to signal a breakpoint. The AMX Breakpoint Manager can be used with such debuggers (see Chapter 13.6).

> Warning
>
> Because the occurrence of an NMI interrupt cannot be controlled, the NMI interrupt can occur at any instant, including within critical sections of AMX.
>
> Consequently, the NMI ISP cannot call any AMX service procedures.

4.7 Special Interrupts

Nonconforming Interrupts

In some systems there may be devices which generate interrupts requiring no communication or synchronization with any task in the system. For example, a high-speed scanner can interrupt the processor whenever new data readings are available. The ISP reads the new data and stores it in memory for subsequent use by any module requiring the information. These modules always use the most recently available value as seen in memory. Interrupts such as this do not need to use AMX services. The ISP simply saves the registers it requires for its own use, processes the interrupting device, restores the registers and immediately returns to the point of interruption using an `IRET` instruction.

Since ISPs of this type use the stack in effect at the time of the interrupt, care must be taken to assure that ALL stack sizes, including the AMX Interrupt Stack and Kernel Stack, are increased to meet the needs of the special ISPs.

ISPs of this type are called **nonconforming ISPs**. You must arrange in hardware that all such nonconforming interrupt sources are of higher priority than all conforming AMX Interrupt Service Procedures. If this is not possible, the private Interrupt Service Procedure must execute with interrupts disabled. Note that the AMX clock ISP (see Chapter 5.2) is considered to be a conforming ISP.

Occasional Task Interaction

In some applications, it may be desirable to bypass AMX for all but certain critical interrupts. For example, in a communication system, normal receive interrupts simply insert the received character into an already available input buffer. Transmit interrupts simply transmit the next available character from an output buffer. These interrupts can be serviced very quickly in a nonconforming ISP, bypassing AMX entirely.

However, when the receiver finally has a complete message or when the transmit buffer goes empty, the device service procedure must send a message to a particular task to inform it that a significant event has occurred on the communication line.

The device service procedure must suddenly become a conforming AMX Interrupt Service Procedure. It can do this by restoring all saved registers and calling a conforming ISP root. The ISP root informs AMX that the interrupt has occurred and calls the device's Interrupt Handler which can then send its message to the task.

Note that if interrupts of this type are employed, all AMX and task stacks must meet the requirements of the nonconforming device ISP.

Shared Interrupt Handlers

Occasionally a single Interrupt Handler can be used to service more than one identical device. For example, an Interrupt Handler for an asynchronous serial I/O device (UART) could be used to service the UART for each of several communication lines.

For the Interrupt Handler to be shared, the code must be reentrant. This usually implies that the information unique to each UART (such as the device port address and line status) must be in a data structure accessible by the handler.

A shared Interrupt Handler must therefore be able to determine which of several device dependent data structures it should use to service a particular device interrupt. A pair of custom ISP roots coded in assembly language can be used to solve this problem.

The following example illustrates how little application code must be written to create a shared Interrupt Handler to handle two devices.

```
struct dvcblock {
   int      port;                          /* Device port              */
   int      line;                          /* Logical line number      */
   :
   Other dynamic device parameters
   };

struct dvcblock dcbA = {0xF8, 1};
struct dvcblock dcbB = {0xE8, 2};

void cdecl deviceih(struct dvcblock *dcbp)
{
   :
   Service device specified by pointer dcbp
   :
   }
```

The two device ISP roots are illustrated below. Each of the ISP roots calls the Interrupt Handler *deviceih* passing it a pointer to a unique, public, device dependent data structure (*dcbA* or *dcbB*). Note that most common C compilers add leading underscores to function and variable names declared in C.

Your application must install the pointers to ISP roots *DVCROOT1* and *DVCROOT2* into the processor Interrupt Vector Table prior to enabling interrupt generation by the corresponding devices.

```
            EXTRN   AAINT:FAR           ;AMX interrupt entry
            EXTRN   AAINX:FAR           ;AMX interrupt exit
;
            EXTERN  _dcbA:DWORD         ;Device control block A
            EXTERN  _dcbB:DWORD         ;Device control block B
            EXTERN  _deviceih:FAR       ;Shared Interrupt Handler
;
;
DVC_CODE SEGMENT BYTE 'CODE'
;
            ASSUME  CS:DVC_CODE,DS:NOTHING
;
            PUBLIC  DVCROOT1            ;ISP root for device #1
            PUBLIC  DVCROOT2            ;ISP root for device #2
;
DVCWB1:     DD      _dcbA               ;A(device control block A)
DVCWB2:     DD      _dcbB               ;A(device control block B)
;
DVCROOT1    PROC    FAR                 ;device ISP root #1
            CALL    AAINT               ;let AMX know
            LDS     SI,CS:DVCWB1        ;DS:SI = A(work block #1)
            JMP     SHORT DVCCMN
;
DVCROOT2    LABEL   FAR                 ;device ISP root #1
            CALL    AAINT               ;let AMX know
            LDS     SI,CS:DVCWB2        ;DS:SI = A(work block #2)
DVCCMN:     PUSH    DS
            PUSH    SI
            CALL    _deviceih           ;common Interrupt Handler
            ADD     SP,4                ;remove parameter
            CALL    AAINX               ;let AMX know
            IRET
;
DVC_CODE    ENDS                        ;end of code segment
```

4.8 Vector Table Initialization

The 80x86 processor operating in real mode vectors to an Interrupt Service Procedure (ISP) directly through its Interrupt Vector Table which, in AMX nomenclature, is simply called the AMX Vector Table.

The Vector Table may be located in RAM or ROM as dictated by your hardware configuration. If the Vector Table is in RAM, it can be further characterized as alterable or not. Whether or not AMX is allowed to alter the content of the Vector Table is determined by the setting in your launch parameter when you start AMX.

Alterable Vector Table

The Vector Table must be initialized to provide dispatch access to all ISPs. If the Vector Table is in RAM and is declared alterable, AMX services can be used to initialize the Vector Table. Your application must initialize the Vector Table entry for each interrupt or exception which your application services. You can use the AMX procedure *ajivtw* (*AAIVTW*) during the AMX launch to install any ISP or ISP root (including an NMI ISP or other nonconforming ISPs or exception handlers) into the Vector Table entry of your choice.

If the Vector Table is in RAM and is declared alterable, AMX will automatically install its own exception handlers for the error exceptions which AMX treats as task traps.

Unalterable Vector Table

Special consideration is required if the Vector Table is to be in ROM or is declared by your launch parameter to be unalterable by AMX.

The Vector Table must be initialized when the ROM image is created. If the Vector Table is in RAM but has been declared unalterable, the RAM copy of the Vector Table must be initialized (as for a ROM image) before AMX is launched. Pointers to all ISPs (including conforming ISP roots, nonconforming ISPs and the NMI ISP) must be installed in the ROM image.

AMX requires that the entries for the error exceptions for which it is responsible be initialized to branch to the addresses specified below.

Vector Number	AMX Entry Point	Purpose
0	*AADERH*	Divide by zero
4	*AAOVFH*	Overflow
5	*AABERH*	Bound error

5. AMX Timing Control

5.1 Introduction to Timing Facilities

Most real-time systems are characterized by the need to provide precise control over the timing of activities. A hardware clock provides the timing source; AMX provides the control over its use.

The unit of time in an AMX system is the system tick which is a fixed interval derived from the hardware clock. The system tick interval is user selectable. Typically, it is set at 10 ms or 100 ms. The system tick interval is chosen to provide the minimum resolution required in a particular application without inflicting unnecessary timing overhead.

Task Delays and Timeouts

A task can suspend itself for a specific interval. A task can also wait for an event which must occur within a specific interval. If the event fails to occur within the interval, the task resumes execution with a timeout indication.

Interval Timers

Application interval timers are the most general form of timer provided by AMX. Once such a timer has been created, it can be started, interrogated and stopped by any task or Interrupt Service Procedure. When a timer is created, an application dependent Timer Procedure must be provided. Whenever the timer expires, AMX executes this Timer Procedure passing it a parameter which was specified when the timer was created.

Application timers can also be periodic. The timer period is specified when the timer is created. AMX calls the corresponding Timer Procedure periodically at the defined interval.

The following AMX procedures provide interval timer services:

ajtmcnv	*AATMCNV*	Convert milliseconds to system ticks
ajtmcre	*AATMCRE*	Create an interval timer
ajtmdel	*AATMDEL*	Delete an interval timer
ajtmrd	*AATMRD*	Read an interval timer
ajtmtag	*AATMTAG*	Find timer id of timer with a specific tag
ajtmwr	*AATMWR*	Start/stop an interval timer

Calendar Clock

The AMX Time/Date Manager provides Y2K compliant time of day calendar support if required. The AMX calendar clock includes second, minute, hour, day, month, year and day of the week. AMX services are provided to set and read the calendar clock. A formatting procedure is also provided to translate the calendar time and date from the internal format in which it is maintained by AMX into an ASCII string in several of the most popular formats.

An application procedure can be tied to the calendar clock and called at one second intervals to permit simple time of day event scheduling.

The following AMX procedures provide calendar clock services:

ajtds	*AATDS*	Set calendar time/date
ajtdg	*AATDG*	Read calendar time/date
ajtdf	*AATDF*	Format calendar time/date as an ASCII string

Time Slicing

The AMX Timer Manager can provide time slicing if required. Time slicing is the process whereby AMX can force tasks which have the same execution priority to share the use of the processor on a round robin basis. The processing interval allocated to a task is called its time slice. The time slice for each task can be uniquely defined permitting fine tuning of the AMX time slice facility to meet each application's particular needs.

The following AMX procedures provide time slicing services:

ajtslv	*AATSLV*	Change a task's time slice interval
ajtsof	*AATSOF*	Disable time slicing
ajtson	*AATSON*	Enable time slicing

5.2 AMX Clock Handler and Kernel Task

AMX includes a conforming clock ISP root, a Clock Handler and a Kernel Task to provide timing facilities. Whenever a clock interrupt occurs, the clock ISP root calls your application clock Interrupt Handler to dismiss the hardware clock interrupt. The ISP root then calls the AMX Clock Handler to trigger the AMX Kernel Task if required. The Kernel Task is triggered at the defined system tick interval if, and only if, there is any outstanding timing activity required in the system. In this case, the interrupted task is suspended and the AMX Kernel Task begins execution.

The AMX Kernel Task monitors all tasks which are in a timed wait state. The timer used by AMX for task waits is maintained privately by AMX for each task. If the timer expires, the Kernel Task removes the task from the wait state. The task is allowed to resume execution when the Kernel Task ends with an indication that a timeout occurred.

The AMX Kernel Task also services all expiring application interval timers. Whenever an interval timer expires, the corresponding Timer Procedure is executed. This procedure can invoke a subset of the AMX services to trigger tasks, send messages to task mailboxes or message exchanges, signal events or wake tasks. If the timer is defined to be periodic, the AMX Kernel Task automatically restarts it with its predefined period.

Once all expiring task timers and application interval timers have been serviced, the AMX Kernel Task ends execution.

Your clock Interrupt Handler can be coded in either C or assembler. For efficiency, assembly language is recommended since most handlers require only one or two machine instructions to dismiss the clock interrupt request.

You must also start your hardware clock at the correct frequency when AMX is launched. You can do this in a Restart Procedure or in some task which is triggered at launch time.

> Note
>
> AMX is delivered to you with clock drivers ready for use with one or more of the timing devices commonly used with the 80x86 processor. These clock drivers are installed as described Appendix C of the AMX 86 Tool Guide.

Clock ISP Implementation

AMX can provide timing facilities only if a hardware clock interrupt is available. For efficiency, it is recommended that the clock ISP be coded in assembly language as illustrated below. Note that the clock ISP root and Interrupt Handler have been merged, making the clock ISP a single body of code. It is assumed in this example that interrupt number 33 has been allocated for clock use.

```
         EXTRN    AACLK:FAR          ;AMX Clock Handler
         EXTRN    AAIVTW:FAR         ;AMX install ISP pointer
;
;
; Assume clock interrupt vector is located in Interrupt Vector Table
; for interrupt number 33
;
CLK_CODE SEGMENT BYTE 'CODE'
;
; The device ISP is located in user program memory
;
         ASSUME   CS:CLK_CODE
;
CLKISP   PROC     FAR
         PUSH     DX                 ;save some registers
         :
         :
         Code, if required, to
         keep the hardware clock running.
         Use only those registers saved.
         Do not enable interrupts.
         :
         :
         POP      DX                 ;restore registers
         CALL     AACLK              ;AMX Clock Handler
         IRET                        ;return from interrupt
;
CLKISP   ENDP
;
;

; Clock Restart Procedure
;
         PUBLIC   RRCLK
;
RRCLK    PROC     FAR
         MOV      AX,SEG CLKISP
         MOV      ES,AX
         MOV      BX,OFFSET CLKISP   ;ES:BX = A(Clock ISP)
         MOV      DX,33              ;clock interrupt number
         CALL     AAIVTW             ;install clock ISP
         RET
;
RRCLK    ENDP
;
CLK_CODE ENDS
```

In this example, the clock ISP pointer (*CLKISP*) must be installed into vector number 33 in the processor Interrupt Vector Table (IVT). AMX service procedure *AAIVTW* has been used in a Restart Procedure for this purpose. When the clock interrupt occurs, the processor disables the interrupt facility, saves the flags and the return address on the interrupted task's stack and fetches the ISP pointer from vector 33 in the IVT. The processor then jumps to the clock ISP at address *CLKISP* in program memory.

The clock ISP must ensure that the clock keeps running. The manner in which this is accomplished is, of course, hardware dependent. Sometimes one or more processor registers will be required in order to access the hardware clock to keep it running. The ISP must save the registers it requires, keep the clock running and restore the registers. The ISP calls the AMX Clock Handler at its entry point *AACLK*.

Note that interrupts must be disabled upon entry to procedure *AACLK*.

If more than one device shares the interrupt, the ISP must determine the interrupt source by testing device status registers. If the clock caused the interrupt, the ISP must dismiss the clock interrupt and keep the clock running. The ISP must then call the AMX Clock Handler at its entry point *AACLK*.

Note that the clock ISP does not call the AMX Interrupt Supervisor entry or exit procedures *AAINT* or *AAINX*. The AMX Clock Handler *AACLK* will invoke the AMX Interrupt Supervisor if, and only if, an AMX clock tick must be serviced by the AMX Kernel.

It is important to note that, because of task switching invoked by function *AACLK*, there may be significant delay before AMX returns from the *AACLK* call and executes the *IRET* instruction to resume execution of the preempted task.

It is preferred that the AMX Clock Handler be called from a clock ISP coded in assembly language as in the example already provided. However, it is possible to code the clock ISP in C as indicated in the following example. This C solution is not recommended since it significantly increases the service overhead for every clock interrupt.

The clock ISP and Restart Procedure can be written as Large or Medium model C functions without formal parameters.

```
#include "amx831cf.h"                    /* AMX C Interface Header   */


static struct amxisps isproot;           /* Clock ISP root           */

void cdecl clockih(void)                 /* Clock Interrupt Handler  */
{
   :
   Dismiss clock interrupt and keep clock running
   Do not enable interrupts.
   :
   ajclk();                              /* AMX Clock Handler        */
   }


void cdecl rrclk(void)                   /* Restart Procedure        */
{
   ajispm(clockih, &isproot);            /* Create Clock ISP root    */
   ajivtw(33, (void (*)())&isproot);     /* Install Clock ISP        */
   }
```

5.3 Interval Timers and Timer Procedures

AMX supports any number of application interval timers in a system. The maximum number in a system is defined in your System Configuration Module (see Chapter 14.5).

A timer must be created by an application before it can be used. Restart Procedures, tasks, ISPs and Timer Procedures can create timers. It is recommended that only Restart Procedures and tasks be used to create timers.

AMX procedure *ajtmcre* is used to create a timer. AMX allocates a timer and returns a timer id to the caller. The timer id is a handle which uniquely identifies the particular timer allocated for use by the application. It is the responsibility of the application to keep track of the timer id for future reference to the timer.

When a timer is created, you can provide a unique 4-character tag to identify the timer. The tag can be used subsequently in a call to *ajtmtag* to find the timer id allocated by the Timer Manager to the particular timer.

When a timer is created, the caller must also specify the following parameters: the timer period, a pointer to an application Timer Procedure and an optional 32-bit application dependent parameter.

The timer period determines if the timer is periodic. If the timer period is zero, the timer is a one-shot timer. Whenever a one-shot timer is started, it runs until it expires at which time it remains idle until started again.

If the timer period is non-zero, the timer is periodic. The period specifies the timer's period measured in AMX system ticks.

Whenever a timer expires, the AMX Kernel Task executes the Timer Procedure which was provided when the timer was created. The Timer Procedure receives the timer id and the predefined 32-bit application parameter as parameters.

When a timer is created, AMX sets the timer idle. The timer remains idle until it is started by a Restart Procedure, task, ISP or Timer Procedure. Timers are started by calling AMX procedure *ajtmwr* to write the initial timer interval into the timer.

Timer intervals are measured in multiples of system ticks. For convenience, the AMX procedure *ajtmcnv* can be used to convert a period specified in milliseconds to the corresponding number of system ticks.

Timers are down-counters. When a timer expires, the AMX Kernel Task calls the timer's Timer Procedure. One-shot timers remain expired unless they are restarted by their Timer Procedure. Periodic timers are automatically restarted by the AMX Kernel Task with their predefined timer period before the timer's Timer Procedure has been executed.

When an interval timer is no longer needed, it can be deleted with a call to *ajtmdel*.

The AMX service procedures which can be called from a Timer Procedure include the following:

ajtrig	*AATRIG*	Start (trigger) a task with no message
ajsend	*AASEND*	Start a task by sending it a message at one of 4 priorities
ajwake	*AAWAKE*	Wake a task which is waiting
ajsgnl	*AASGNL*	Signal a task
ajtmcre	*AATMCRE*	Create an interval timer
ajtmdel	*AATMDEL*	Delete an interval timer
ajtmrd	*AATMRD*	Read an interval timer
ajtmwr	*AATMWR*	Start/stop an interval timer
ajtmcnv	*AATMCNV*	Convert milliseconds to system ticks
ajsmsig	*AASMSIG*	Signal to a semaphore
ajevsig	*AAEVSIG*	Signal one or more events in an event group
ajmxsnd	*AAMXSND*	Send message to message exchange
ajbget	*AABGET*	Get a buffer from a specific pool
ajbfre	*AABFRE*	Free a buffer
		Circular List Manager services
		Linked List Manager services

The following conditions exist when the Timer Procedure is called by the AMX Kernel Task:

Interrupts are enabled.
All registers are free for use.

DS,ES	*DGROUP* segment
SS:SP	AMX Kernel Stack ready for use
BP	Offset of parameters on stack

The direction flag is set to forward.

Timer Procedures can be written as Large or Medium model C functions with formal parameters.

```
#include "amx831cf.h"                    /* AMX C Interface Header   */

void cdecl truser(                       /* Timer Procedure          */
AMXID timerid,                           /* timer id                 */
struct userblock FAR *userp)             /* pointer to user block    */
{
   :
   Do timer expiry processing
   :
   }
```

Timerid is the timer id assigned by AMX to the interval timer when it was created. *Userp* is the 32-bit application parameter provided when the timer was created. In this example, it is assumed that *userp* is a *FAR* pointer to an application structure of type *userblock*.

The Timer Procedure must execute with the interrupt facility enabled. If interrupts must be temporarily disabled, they must be enabled prior to returning to the AMX Kernel Task. A Timer Procedure must not issue any AMX directives which would in any way force the Kernel Task to wait for any reason.

> Note
>
> Timer Procedures must only use AMX services which are marked in Chapter 16 as
> ■ Timer Procedure

Application Timer Procedures use the AMX Kernel Stack.

In addition to the minimum stack size required for the AMX Kernel Stack, you must allocate sufficient stack to satisfy the worst case requirements of all application Timer Procedures.

The Timer Procedure can be coded in assembler as a *FAR* procedure as follows:

```
TMR_CODE SEGMENT BYTE 'CODE'
;
; The Timer Procedure is located in user program memory
;
         ASSUME   CS:TMR_CODE
;
TRUSER   PROC     FAR
         MOV      DX,WORD PTR [BP]     ;DX = timer id
         LES      BX,DWORD PTR [BP+2]  ;ES:BX = 32-bit timer parameter
         :
         :
         Timer Procedure code goes here
         :
         :
         RET
;
TRUSER   ENDP
;
TMR_CODE ENDS
```

5.4 Task Time Slicing

AMX provides task time slicing as an option. The AMX system must be configured to include a clock if time slicing is to be possible. Time slice intervals are then specified as multiples of the AMX system tick.

Time slicing is normally disabled. It is enabled with a call to AMX procedure *ajtson*. It can be disabled again with a subsequent call to *ajtsof*.

If your AMX System Configuration Module indicates that time slicing is required, AMX will automatically enable time slicing by calling *ajtson* prior to executing any application Restart Procedure at launch time.

Time slicing is a feature which coexists with the normal AMX preemptive priority-based task scheduling. Whether or not a task is time-sliced depends on two things: the task's execution priority and the task's time slice interval. Both of these parameters are first defined when a task is created.

If a task's time slice interval is zero, the task will not be time sliced. A non-zero time slice interval specifies the number of AMX system ticks which will be given to the task before the task is forced to relinquish the processor to another time sliced task.

A task is time sliced with all other tasks having the same execution priority and a non-zero time slice interval. A set of time sliced tasks are normally created to share a particular execution priority. Tasks which are not time sliced (their time slice interval is zero) are usually assigned to different execution priority levels. More than one set of time sliced tasks can be created, each set existing at a different priority level. Other tasks can reside at priorities between the time sliced sets.

The order in which tasks at a particular shared priority level are given their time slice by AMX is determined by the chronological order in which the tasks were created.

Figure 5.4-1 illustrates the allocation of processing time to two tasks, B and C. Task B was created first with a time slice interval of 100 AMX system ticks. Task C was created later with a time slice interval of 50 AMX system ticks. At time *t1*, tasks B and C were both triggered by a higher priority task A which then ended.

Since task B was created first, it executes first. Thereafter, in the absence of any other higher priority task activity, the processor is shared by tasks B and C as illustrated.

Task C begins to execute at time *t2* when task B's first 100 tick time slice expires. At time *t3*, task B ends execution. The processor is immediately given to task C which continues to execute without further interruption by task B.

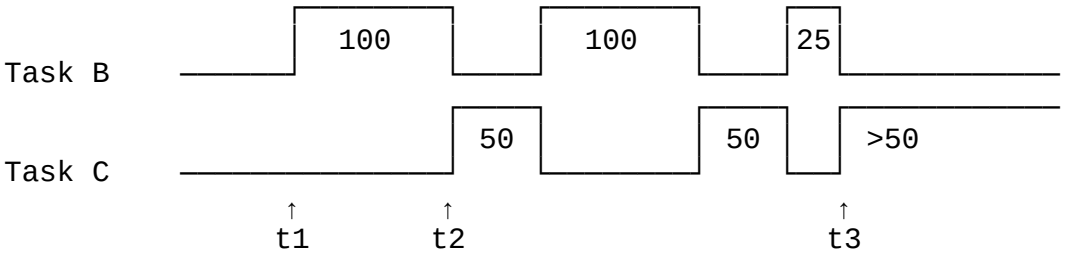


Figure 5.4-1 Simple Time Slicing

Figure 5.4-2 illustrates the effects to be expected if, while tasks B and C are being time sliced, higher priority task A is invoked and executes to completion within 125 system ticks.

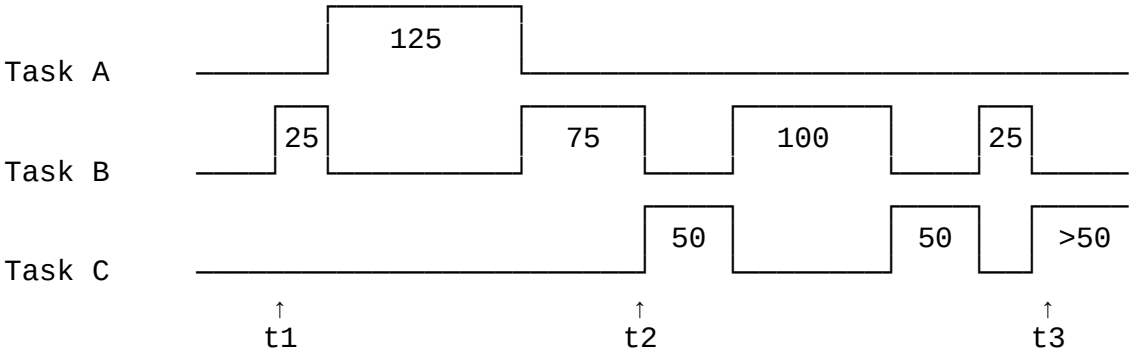


Figure 5.4-2 Interrupted Time Slicing

Time slicing can be completely disabled at any time with a call to AMX procedure *ajtsof*. Procedure *ajtson* can then be used to enable time slicing.

Task time slice intervals can be dynamically adjusted using AMX procedure *ajtslv* to fine tune the shared use of the processor based on observed effects.

The starting and ending of time sliced tasks are not synchronized to the AMX clock. However, switching between time sliced tasks will only occur at AMX system ticks because the switch is initiated by the AMX Kernel Task.

It should be noted that if higher priority tasks frequently interrupt a time sliced task, the exact processing time allocated to the task will not exactly equal *n*t* where *n* is the task's time slice interval and *t* is the AMX system tick period. The task will be forced to relinquish the processor after the *n*'th system tick which occurs while the task is executing. However, because of higher priority task activity, the task did not get the use of the processor for the entirety of each of its *n* system ticks.

Finally, if a task is time sliced and all other tasks at the same execution priority are idle or blocked, a time slicing overhead penalty will still exist. The AMX Kernel Task will continue to interrupt the task at its time slice interval, conclude that task switching is unnecessary and resume execution of the interrupted task.

Suggestion

Do not use time slicing unless its use is warranted. Better use of processor time results if tasks are allowed to run to completion.

Misuse is abuse!

5.5 Time/Date Manager

Most real-time systems require the maintenance of a calendar clock. The AMX Time/Date Manager provides this facility.

The Y2K compliant calendar clock maintained by the Time/Date Manager includes second, minute, hour, day, month and year. Leap year is accounted for. The day of the week (Mon to Sun) is also maintained.

An application Scheduling Procedure can be connected to the calendar clock to provide activity scheduling based on the time of day.

The Time/Date Manager includes a set of service procedures for use by application tasks, Timer Procedures, Interrupt Service Procedures and Restart Procedures. These include:

ajtdg	*AATDG*	Get Time and Date
ajtds	*AATDS*	Set Time and Date
ajtdf	*AATDF*	Format Time and Date as an ASCII string

Several conflicts always arise with the maintenance of time and date. These conflicts arise when:

1) the clock is trying to update the time and date while
2) a task is trying to read the time and date and
3) another task is trying to set the time and date.

The conflicts are even more pronounced if separate calls are required to get both time and date. The Time/Date Manager resolves these conflicts.

Operation

The Time/Date Manager includes two components: an AMX Restart Procedure and a set of service procedures.

If your AMX System Configuration Module enables the Time/Date option, AMX automatically calls the Time/Date Restart Procedure during the launch prior to executing any application Restart Procedure. The calendar clock is set to 00:00:00 Monday October 01/1990 (the distribution date of the Time/Date Manager).

The Time/Date Manager creates a periodic interval timer and starts it with a one second period. At one second intervals thereafter, the AMX Kernel Task executes the Time/Date Timer Procedure. The current time and date are updated by one second.

The Time/Date Manager employs an interlock mechanism to ensure that any race between a task trying to set a new time and date and the Time/Date Timer Procedure trying to update time and date is resolved. The task's new time and date have precedence.

This interlock mechanism also assures that any task trying to read the current time and date does not get a partially updated (and hence erroneous) time and date.

Once the time and date have been updated, the Time/Date Timer Procedure calls an application Scheduling Procedure if one has been provided. This procedure, called once a second, is thus tied to the calendar clock and can be used to initiate activities which must be time of day driven. A detailed description of this facility is provided later in this chapter.

The Time/Date Timer Procedure ends once the application Scheduling Procedure (if any) has been executed.

Time/Date Structure

The Time/Date Manager provides time and date in the following form. C structure *amxtds* is defined in the AMX header file *AMX831SD.H*. Assembler structure *AMXTDS* is defined in the AMX definition file *AMX831SD.DEF*.

The C and assembler structure definitions are shown below.

```
/* AMX Time/Date Structure                                          */

struct amxtds {

   unsigned char amtdsec;        /* seconds  (0-59)                 */
   unsigned char amtdmin;        /* minutes  (0-59)                 */
   unsigned char amtdhr;         /* hours    (0-23)                 */
   unsigned char amtddy;         /* day      (1-31)                 */
   unsigned char amtdmn;         /* month    (1-12)                 */
   unsigned char amtdyr;         /* year     (0-99)                 */
   unsigned char amtdow;         /* day of week (Mon=1 to Sun=7)    */
   unsigned char amtdcen;        /* 0 if time/date is incorrect     */
                                 /* century if time/date is correct */
   };


; AMX Time/Date Structure
;
AMXTDS    STRUC
;
AMTDSEC   DB      ?           ;seconds    (0-59)
AMTDMIN   DB      ?           ;minutes    (0-59)
AMTDHR    DB      ?           ;hours      (0-23)
AMTDDY    DB      ?           ;day        (1-31)
AMTDMN    DB      ?           ;month      (1-12)
AMTDYR    DB      ?           ;year       (0-99)
AMTDOW    DB      ?           ;day of week (Mon=1 to Sun=7)
AMTDCEN   DB      ?           ;0 if time/date is incorrect
;                             ;century if time/date is correct
;
AMXTDS    ENDS
;
```

Time/Date Validity

The century is used as follows. At startup, the Time/Date Restart Procedure resets the century to 0 to indicate that the initial default time and date are incorrect. Note that the initial time and date are valid; they are just not correct. At some later time, an application program can issue a call to the Time/Date service procedure *ajtds* to set a correct time and date. The century specified as a parameter in that call should be set indicating that the time and date parameters are correct.

The century can be reset to 0 by an application program with an *ajtds* call to indicate that the time and date are incorrect.

When setting the time and date, the day of the week does not have to be provided as long as the year lies between 1974 and 2099. The Time/Date Manager will figure it out and set it accordingly.

Time/Date Scheduling

Many real-time systems require that certain activities be performed at specific times of day. For instance, it may be required that every day at 8:00 a.m. a 24 hour report be generated. Or maybe every 1/2 hour, measured from the hour, the system must make a measurement and display the result. The Time/Date Manager provides the mechanism necessary to implement such features.

At one second intervals the Time/Date Timer Procedure updates the time and date and then calls an application Scheduling Procedure. The name of this procedure must be provided in your System Configuration Module (see Chapter 14.4).

Your Scheduling Procedure is called with a *FAR* procedure call with an actual Time/Date structure as a parameter. Note that the structure is passed to the procedure by value. The structure specifies the time and date at the instant the procedure is called.

The Scheduling Procedure is application dependent. It executes as part of the Time/Date Timer Procedure. It must therefore not be compute or I/O bound.

In general, the procedure checks the time (and date if necessary) and determines if some application dependent action must be initiated at that instant. If action is required, the procedure either performs the action directly or requests AMX to start some other task which is responsible for taking the required action.

You must be sure to allocate sufficient stack to the AMX Kernel Task to accommodate the needs of the Scheduling Procedure.

The following conditions exist when your Time/Date Scheduling Procedure is called by the Time/Date Manager

Interrupts are enabled.
All registers are free for use.

DS,ES	*DGROUP* segment
SS:SP	AMX Kernel Stack ready for use
BP	Offset of parameters on stack

The direction flag is set to forward.

The Scheduling Procedure can be written as a Large or Medium model C procedure as follows:

```
#include "amx831sd.h"                  /* AMX Structure Definitions   */

void cdecl tdshed(                     /* Scheduling Procedure        */
int dummy)                             /* Dummy parameter             */
{
   struct amxtds *ptdp;                /* Pointer to time/date        */


   ptdp = (struct amxtds *)&dummy;
   :
   Perform required tests and initiate
   actions if it is time for them
   :
   }
```

Note that function *tdshed* is declared to receive a dummy integer parameter. The function actually receives an entire *amxtds* time/date structure passed by value. However, not all C compilers support this facility. Therefore, we recommend that you use a local pointer variable *ptdp* to gain access to the time/date structure as illustrated above.

Your Time/Date Scheduling Procedure can be coded in assembly language as a *FAR* procedure as follows:

```
         INCLUDE  AMX831SD.DEF            ;AMX Structure Definitions
;
;
TDS_CODE SEGMENT BYTE 'CODE'
;
; Time/Date Scheduling Procedure located in program memory
;
         ASSUME   CS:TDS_CODE
;
         PUBLIC   TDSHD
;
TDSHD    PROC     FAR
         MOV      AL,[BP].AMTDSEC         ;AL = seconds
         :
         MOV      DL,[BP].AMTDDY          ;DL = day
         :
         :
         Perform required tests and initiate
         actions if it is time for them
         :
         :
         RET
;
TDSHD    ENDP
;
TDS_CODE ENDS
```

Note that function *TDSHD* receives an entire *AMXTDS* time/date structure passed by value on its stack. Since register *BP* is initialized with the offset of the structure on the stack, register *BP* can be used to directly access the time/date structure as illustrated above.

Time/Date ASCII Formats

The Time/Date Manager procedure *ajtdf* can be used to format time and date into an ASCII character string in any of several popular formats. The time and date is presented to *ajtdf* in the standard AMX time/date structure. The ASCII string is returned in a character buffer provided by the caller. See Chapter 16 for the *ajtdf* calling sequence.

Time can be formatted as either of:

```
HH:MMb
HH:MM:SSb
```

where:	*HH*	= hours	(00 to 23)	
	MM	= minutes	(00 to 59)	
	SS	= seconds	(00 to 59)	
	b	= space	time/date are correct	(century <> 0)
	b	= #	time/date are incorrect	(century = 0)

The date can be formatted as any of:

```
DDD MMM dd/ccyyb
DDD mm/dd/ccyyb
DDD dd MMM/ccyyb
DDD dd/mm/ccyyb
DDD ccyy/mm/ddb
```

where:	*DDD*	= day of week	(Mon to Sun)
	MMM	= month	(Jan to Dec)
	mm	= month	(01 to 12)
	dd	= day	(01 to 31)
	cc	= century	(19 or 20)
	yy	= year	(00 to 99)
	b	= space	

The 4 character day-of-week (*DDDb*) and/or the two character century (*cc*) can be omitted.

The format of time and/or date is specified by a format specification parameter, a single byte presented to procedure *ajtdf*.

Figure 5.5-1 describes the format specification byte and the effect of each bit in it on the formatting of the time and date.

7	6	5	4	3	2	1	0
TD1	*TD0*	*C*	*M*	*S*	*W*	*D1*	*D0*

TD1	*TD0*	Time/Date selection	
0	0	time followed by date	23:59:59 Sun Jan 31/93
0	1	time only	23:59:59
1	0	date only	Sun Jan 31/93
1	1	date followed by time	Sun Jan 31/93 23:59:59

S = 0	display seconds	23:59:59
S = 1	suppress seconds	23:59

W = 0	suppress day of week	23:59:59 Jan 31/93
W = 1	display day of week	23:59:59 Sun Jan 31/93

M	*D1*	*D0*	Date format	
0	0	0	American alphanumeric form	Jan 31/93
0	0	1	American numeric form	01/31/93
0	1	0	European alphanumeric form	31 Jan/93
0	1	1	European numeric form	31/01/93
1	0	0	Metric form	93/01/31

C = 0	suppress century	Jan 31/93
C = 1	include century	Jan 31/1993

Figure 5.5-1 Time/Date Format Specification Parameter

This page left blank intentionally.

6. AMX Semaphore Manager

6.1 Introduction

E.W. Dijkstra introduced two primitive operations to resolve two seemingly unrelated problems: mutually exclusive access by tasks to critical resources and the synchronization of asynchronously occurring activities.

The abstract primitives, called P and V operators, operate on a variable called a semaphore. Many variations of these P and V operators have been implemented since their first introduction. The AMX Semaphore Manager provides two variations: a counting semaphore which has been enhanced to provide priority queuing and automatic timeout, and a resource semaphore in which resource ownership is tied to a specific task.

A counting semaphore is a semaphore with an associated counter which is incremented by the P operator (signal) and decremented by the V operator (wait). The resource (object) controlled by the semaphore is free (available) when the counter is greater than 0. The number of resources (objects) controlled by the semaphore is determined by the maximum value the counter is allowed to achieve.

A **counting semaphore** with a maximum count of one is a binary semaphore. It can be used to provide mutually exclusive access to a single resource or to signal a particular event.

A **resource semaphore** is a binary semaphore which can only be used for resource ownership control. It differs from a counting semaphore in one significant feature: resource ownership is tied to a specific task. No other task except the task owning the resource is allowed to signal the release of the resource.

The counting and resource semaphores provided by the Semaphore Manager have been enhanced to provide priority queuing and automatic timeout. Tasks which wait on semaphores can specify the priority at which they wish to wait for the resource or event controlled by the semaphore. Tasks can also specify the maximum interval which they are prepared to wait.

The AMX Semaphore Manager provides the following semaphore management services:

ajsmcre	*AASMCRE*	Create a semaphore
ajsmdel	*AASMDEL*	Delete a semaphore
ajsmrsv	*AASMRSV*	Reserve a resource semaphore (optional timeout)
ajsmrls	*AASMRLS*	Release a resource semaphore (nested)
ajsmfre	*AASMFRE*	Free a resource semaphore (unconditional)
ajsmwat	*AASMWAT*	Wait on a counting semaphore (optional timeout)
ajsmsig	*AASMSIG*	Signal to a counting semaphore
ajsmget	*AASMGET*	Get use of a counting semaphore (no wait)
ajsmtag	*AASMTAG*	Find the semaphore id of a semaphore with a specific tag

Your use of the Semaphore Manager is optional. If you intend to use it, you must indicate so in your System Configuration Module. You must also provide a hardware clock and include the AMX timing facilities.

Semaphores can be predefined in your System Configuration Module which is processed by the Semaphore Manager at startup. Semaphores which are predefined are automatically created by the Semaphore Manager. The semaphore id assigned to each predefined semaphore is stored in a variable which you must provide for that purpose.

6.2 Semaphore Use

The Semaphore Manager supports any number of semaphores. The maximum number of semaphores in a system is defined in your System Configuration Module (see Chapter 14.5). The defined maximum sets an upper limit on the number of actual semaphores that can be created in your application.

A semaphore must be created by an application before it can be used. Restart Procedures, tasks, ISPs and Timer Procedures can create semaphores. It is recommended that only Restart Procedures and tasks be used to create semaphores.

A semaphore is created with a call to procedure *ajsmcre* indicating the type of semaphore, counting or resource, to be created. The Semaphore Manager allocates a semaphore and returns a semaphore id to the caller. The semaphore id is a handle which uniquely identifies the semaphore. It is the responsibility of the application to keep track of the semaphore id for future reference to the semaphore.

When a semaphore is created, you can provide a unique 4-character tag to identify the semaphore. The tag can be used subsequently in a call to *ajsmtag* to find the semaphore id allocated by the Semaphore Manager to the particular semaphore.

When a semaphore is no longer needed, it can be deleted with a call to *ajsmdel*. The Semaphore Manager will reject the attempt to delete the semaphore if any task is waiting for the use of the semaphore. When the Semaphore Manager deletes a semaphore, it marks the semaphore as invalid such that any subsequent reference to the semaphore will be rejected.

You must be absolutely certain that no task, ISP or Timer Procedure is referencing the semaphore just as you go to delete it. Be aware that the deleted semaphore id may immediately be reused by AMX for some other purpose.

Counting Semaphore

A counting semaphore is created by specifying an initial semaphore count greater than or equal to zero in the call to *ajsmcre*. When used for mutual exclusion, the semaphore should be given an initial value of one.

If a semaphore is initialized with a semaphore value of *n*, it can be used to control access to *n* resources of a particular type. For instance, a counting semaphore with an initial count of three could be used to control access to three printers in a system. When used in this fashion, the Semaphore Manager assures that no more than three tasks can own a printer at any one time. However, the Semaphore Manager does not provide any guidance as to which of the three available printers a task can use.

Access to the resource controlled by a counting semaphore is acquired with a call to procedure *ajsmwat*. Only tasks are allowed to call this procedure because the caller may have to wait for the resource to become available. When requesting use of a resource, the task specifies the interval, measured in system ticks, which it is willing to wait for the resource if the resource is unavailable at the time of the call. The task also specifies the priority at which it is prepared to wait, zero being the highest priority.

If the resource controlled by the semaphore is available, the Semaphore Manager gives it to the calling task immediately. If the resource is unavailable, the Semaphore Manager will add the task to the semaphore's wait queue at the priority it said it was willing to wait. Thus tasks which require high priority access to the resource can preempt lower priority waiting tasks in the wait queue.

When the task is finished using the resource, it signals its release by calling procedure *ajsmsig*. The Semaphore Manager releases the resource and checks the semaphore's wait queue. If any tasks are waiting on the queue, the resource is immediately given to the task at the head of the wait queue. If that task is of higher priority than the task which is releasing the resource, a task switch will occur giving the higher priority task an immediate opportunity to use the resource. If the task being granted use of the resource is of lower priority than the task which is releasing it, the new owner will have to wait until the currently executing task relinquishes control of the processor.

If the semaphore does not become available within the timeout interval specified by the task, the task will be removed from the semaphore wait queue and will resume execution with a timeout indication.

Tasks, ISPs or Timer Procedures which need to use a resource but which cannot wait for the resource must call *ajsmget* to get use of the resource. Unlike *ajsmwat*, procedure *ajsmget* will not allow the caller to wait on the semaphore queue. If the resource cannot be immediately granted to the caller, *ajsmget* returns an error indication.

Resource Semaphore

AMX resource semaphores provide the simplest mechanism for controlling the access to critical resources. Resources may include disk files, I/O devices, database components, regions of memory, specific words of memory or any other entity which is considered to be a resource.

An application task requests ownership of a resource with a call to the Semaphore Manager. If the resource is available, the task is granted immediate ownership of it. If the resource is unavailable, the task is inserted on a list of tasks waiting for the resource at a wait priority specified by the task. The task can optionally specify the maximum time interval it is prepared to wait for access. When the task which currently owns the resource releases it with a call to the Semaphore Manager, the resource will be given to the task with the highest waiting priority.

Although a resource semaphore uses a binary semaphore for controlling access to the resource, it differs from the general counting semaphore facility provided by the AMX Semaphore Manager in one significant feature. Resource ownership is tied to a specific task. Only the task owning such a resource is permitted to signal its release. The Semaphore Manager does not permit more than one task to share ownership of such a resource.

A resource semaphore is created by specifying an initial semaphore count of -1 in the call to *ajsmcre*. The Semaphore Manager creates a resource semaphore and automatically gives it an initial value of one indicating that the resource is free.

A resource is reserved by calling procedure *ajsmrsv* using the semaphore id allocated to the particular resource semaphore when it was created. Only tasks can reserve a resource controlled by a resource semaphore. The task which owns the resource can reserve it again and again resulting in a nested reservation.

When requesting use of a resource, the task specifies the interval, measured in system ticks, which it is willing to wait for the resource if the resource is unavailable at the time of the call. The task also specifies the priority at which it is prepared to wait, zero being the highest priority.

The resource is released with a call to *ajsmrls*. The task which owns the resource must release the resource once for each nested reserve that it has made. The resource becomes free when the nesting count reaches zero. Alternatively, the resource can be released unconditionally with a call to *ajsmfre*.

If the resource is in use when *ajsmrsv* is called, the task will be added to the resource semaphore queue forcing the task to wait at the priority specified by the task in its call to reserve the resource. When the current owner of the resource releases it, the Semaphore Manager gives the resource to the task (if any) waiting at the beginning of the resource semaphore queue. Hence, a task must wait until all other tasks ahead of it in the queue use and release the resource.

If the task is of higher priority than the task which is releasing the resource, a task switch will occur giving the higher priority task an immediate opportunity to use the resource. If the task being granted use of the resource is of lower priority than the task which is releasing it, the new owner will have to wait until the currently executing task relinquishes control of the processor.

If the resource does not become available within the timeout interval specified by the task, the task will be removed from the resource semaphore wait queue and will resume execution with a timeout indication.

Tasks which need to use a resource but which cannot wait for the resource to be free can still call *ajsmrsv*. However, they must specify a timeout value of less than zero so that they will not wait on the resource semaphore queue. If the resource cannot be immediately granted to the caller, *ajsmrsv* returns an error indication.

6.3 Semaphore Applications

Mutual Exclusion

Assume that three tasks, A, B and C, require shared access to a common data structure being used to control some process. Access to the data structure must be mutually exclusive so that one task cannot be modifying the data in the structure while another task is accessing it.

A semaphore with an initial count of one is required because there is only one data structure to manage. A counting semaphore is created in a Restart Procedure and initialized with a count of one. One task can own the resource; two tasks can be waiting. Note that the Semaphore Manager does not guarantee that only tasks A, B and C can access the data structure.

In this example, Task A waits indefinitely (timeout value = 0) at priority level 20 for access to the data variable. Task B waits indefinitely at priority level 10. Task C only waits for 100 system ticks at priority 5 before it assumes that it cannot have the resource.

This example is illustrated on the next page. The manner in which Tasks A, B and C are created and started is beyond the scope of this example.

Parameters 1 and 2 in the data structure are modified by the tasks to illustrate that their alteration by one task is guaranteed by the semaphore to be completed before access by either of the other two tasks is allowed.

```
#include "amx831cf.h"              /* AMX C Interface Header         */

static AMXID daccess;              /* Data access semaphore id       */

static struct {
   int      dbpar1;                /* parameter 1                    */
   int      dbpar2;                /* parameter 2                    */
   :
   :
   } datavar;                      /* Data variable                  */


void cdecl rruser(void)            /* Restart Procedure              */
{
   ajsmcre(&daccess, 1, "DACS");   /* Create counting semaphore      */
   }


void cdecl sttaskA(void)           /* Task A                         */
{
                                   /* Wait for access                */

   if (ajsmwat(daccess, 0, 20) == AEROK) {
            datavar.dbpar1 = 1;    /* Set parameters                 */
            datavar.dbpar2 = 1;
            }
   }


void cdecl sttaskB(void)           /* Task B                         */
{
                                   /* Wait for access                */

   if (ajsmwat(daccess, 0, 10) == AEROK) {
            datavar.dbpar1 = 2;    /* Set parameters                 */
            datavar.dbpar2 = 2;
            }
   }


void cdecl sttaskC(void)           /* Task C                         */
{
                                   /* Wait 100 ticks for access      */

   if (ajsmwat(daccess, 100, 5) == AEROK) {
            datavar.dbpar1 = 3;    /* Set parameters                 */
            datavar.dbpar2 = 3;
            }
   }
```

Task/Event Synchronization

A counting semaphore can be used to provide synchronization between a task waiting for an event and a task, ISP or Timer Procedure in which the event is detected. The following example assumes that a device ISP detects the event.

A task creates a counting semaphore with an initial count of zero.

The task then starts the device and waits on the semaphore. When the event of interest is detected by the ISP, it signals to the semaphore. The Semaphore Manager grants access to the waiting task which then resumes execution knowing that the event occurred.

```
#include "amx831cf.h"          /* AMX C Interface Header                */

static AMXID syncisp;          /* ISP synchronization semaphore id      */

void cdecl sttask(void)
{
   int      status;


   ajsmcre(&syncisp, 0, "SISP");    /* Create counting semaphore       */
   :
   Start device I/O
   :
                                    /* Wait 1 second for event         */
   status = ajsmwat(syncisp, ajtmcnv(1000), 0);

   if (status == AEROK) {           /* OK?                             */
            :
            Event occurred within 1 second
            :
            }

   else if (status == AERTMO) {     /* Timeout?                        */
            :
            Event did not occur within 1 second
            :
            }

   else {                           /* Fatal error                     */
            :
            No such semaphore exists (who deleted it?)
            :
            }

   ajsmdel(syncisp);          /* Delete semaphore                      */
   }
```

```
          EXTRN    AAINT:FAR          ;enter ISP
          EXTRN    AAINX:FAR          ;leave ISP
          EXTRN    AASMSIG:FAR        ;signal to a semaphore
          EXTRN    _SYNCISP:WORD      ;ISP synchronization semaphore id
;
;
DVC_CODE SEGMENT BYTE 'CODE'
;
          ASSUME   CS:DVC_CODE
;
;
DVCISP    PROC     FAR
          CALL     AAINT              ;tell AMX
          :
          Clear interrupt source
          Check for event
          :
          MOV      BX,_SYNCISP        ;BX = semaphore id
          CALL     AASMSIG            ;signal to semaphore
          :
          :
          CALL     AAINX              ;tell AMX
          IRET                        ;end of interrupt
;
DVCISP    ENDP
;
DVC_CODE ENDS
```

In this example, we have mixed C and assembly language for illustration purposes. Notice the use of the underscore character in the assembler code reference to variable *_SYNCISP*. In C, the variable is named *syncisp*. The underscore is required because we have assumed that the C compiler actually assigns the symbol name *_syncisp* to the variable. Also note that we have assumed that the program linker can ignore upper/lower case differences in symbols. Your C compiler and linker may behave differently.

Refer to the AMX 86 Tool Guide for a more complete description of C, assembler and linking requirements when using AMX with a particular toolset.

Resource Nesting

Assume that two tasks, A and B, have to share a numeric coprocessor. Furthermore, these two tasks also must share a common library procedure *ncmath* which must use the coprocessor.

A resource semaphore must be used because ownership of the numeric coprocessor must be tied to one task at a time and the owner must be allowed to make nested resource reservation calls.

In the example, Task A waits indefinitely (timeout value = 0) at priority level 20 for the use of the numeric coprocessor. Once the task owns the coprocessor, it initializes it and calls *ncmath* to perform some mathematical operation using it. Task A then finishes using the coprocessor and releases it.

Task B does not use the coprocessor directly. It calls *ncmath* to perform a mathematical operation which requires the use of the coprocessor.

The math library procedure waits forever at priority 20 to reserve the coprocessor without knowing which task is calling. If *ncmath* is called by Task B while Task A owns the coprocessor, Task B will be suspended by the Semaphore Manager until Task A is finished with the coprocessor.

The manner in which Tasks A and B are created and started is beyond the scope of this example.

For simplicity, Tasks A and B do not check the error codes returned by *ncmath* or the Semaphore Manager. In practice, error codes should never be ignored.

```
#include "amx831cf.h"                  /* AMX C Interface Header      */

static AMXID ncaccess;                 /* Coprocessor resource        */
                                       /* semaphore id                */


void cdecl rruser(void)
{
   ajsmcre(&ncaccess, -1, "387P");     /* Create resource semaphore   */
   }


int ncmath(
AMXID ncid)                            /* Semaphore id                */
{
   int      status;

   status = ajsmrsv(ncid, 0, 20);      /* Reserve coprocessor         */
   if (status == AEROK) {
            :
            use coprocessor and do math operation
            :
            status = ajsmrls(ncid);    /* Release coprocessor         */
            }
   return(status);
   }


void cdecl sttaskA(void)
{
   ajsmrsv(ncaccess, 0, 20);           /* Reserve coprocessor         */
   :
   initialize coprocessor
   :
   ncmath(ncaccess);                   /* Common math operation       */
   ajsmrls(ncaccess);                  /* Release coprocessor         */
   :
   }


void cdecl sttaskB(void)
{
   :
   ncmath(ncaccess);                   /* Common math operation       */
   ;
   }
```

7. AMX Event Manager

7.1 Introduction

The AMX Event Manager provides the most general form of event synchronization offered by AMX. The Event Manager provides a convenient mechanism for separating the tasks waiting for events from the tasks, Timer Procedures and Interrupt Service Procedures which can signal the event. The Event Manager also allows more than one task to simultaneously wait for a particular event. Tasks can wait for a particular combination of events or for any one of a set of events to occur.

The Event Manager provides a set of event flags which can be associated with specific events in your system. These event flags are provided in groups with 16 event flags per group. Note that the number of event flags per group is dictated by the basic register width (integer size) of the target processor. Hence, 32-bit versions of AMX have 32 event flags per group.

As an example, suppose that two tasks, A and B, must each wait until a motor turns on. Assume that an interrupt will occur when the motor is turned on.

It is assumed that Tasks A and B are completely independent; their processing is unrelated. One method of accomplishing the necessary synchronization is for both tasks to set software flags indicating that they are waiting for the motor and then call AMX procedure *ajwait* to wait for the motor to turn on. When the motor control ISP detects that the motor has started, it could check the wait flags for the two tasks and wake the tasks if necessary with calls to *ajwake*.

This solution does not provide good functional separation between processes. The motor control ISP must be aware that Tasks A and B are waiting for the motor to turn on. As you can imagine, this lack of functional separation is compounded when there are many types of events occurring in a system.

The Event Manager provides a more general solution to this problem. An event flag is defined to represent the state of the motor (off or on). A Restart Procedure initializes the event flag so that it matches the actual state of the motor. When Tasks A and B must wait for the motor, they do so by calling the Event Manager requesting to wait until the motor control event flag indicates that the motor is on. When the motor control ISP detects that the motor is on, it signals the event with a call to the Event Manager. The Event Manager wakes all tasks, including Tasks A and B, which are waiting for the motor to be on.

The AMX Event Manager provides the following event management services:

ajevcre	*AAEVCRE*	Create an event group
ajevdel	*AAEVDEL*	Delete an event group
ajevsig	*AAEVSIG*	Signal one or more events in a group
ajevwat	*AAEVWAT*	Wait for all/any of a set of events in a group (optional timeout)
ajevrd	*AAEVRD*	Read current state of events in a group
ajevnt	*AAEVNT*	Get state of event flags at completion of event wait
ajevtag	*AAEVTAG*	Find the event group id of an event group with a specific tag

Your use of the Event Manager is optional. If you intend to use it you must indicate so in your System Configuration Module. You must also provide a hardware clock and include the AMX timing facilities.

Event groups can be predefined in your System Configuration Module which is processed by the Event Manager at startup. Event groups which are predefined are automatically created by the Event Manager. The event group id assigned to each predefined event group is stored in a variable which you must provide for that purpose.

7.2 Event Synchronization

The AMX Event Manager supports any number of event groups in a system. Each event group includes 16 event flags. The maximum number of event groups in a system is defined in your System Configuration Module (see Chapter 14.5). The defined maximum sets an upper limit on the number of actual event groups that are available through the Event Manager in your application.

Each event in a group is represented by a boolean flag representing the state of the event. The event flags are represented in one 16-bit variable. It is recommended that the boolean states 0 and 1 be used as false and true indicators respectively. The zero state therefore represents no event. The one state indicates that the event has occurred.

An event group must be created by an application before it can be used. Restart Procedures, tasks, ISPs and Timer Procedures can create event groups. It is recommended that only Restart Procedures and tasks be used for this purpose.

Event Manager procedure *ajevcre* is used to create an event group. The Event Manager allocates an event group and returns a group id to the caller. The group id is a handle which uniquely identifies the particular event group allocated for use by the application. It is the responsibility of the application to keep track of the group id for future reference to the event group.

When an event group is created, you can provide a unique 4-character tag to identify the event group. The tag can be used subsequently in a call to *ajevtag* to find the event group id allocated by the Event Manager to the particular event group.

When an event group is created, the caller specifies the initial state that each of the event flags in the group is to assume. Hence, the process of creating a group automatically initializes all events in the group to a predefined state. The assignment of events to specific event flags in the event group is completely determined by the system designer.

Once an event group has been created, it can be used to synchronize tasks to any of the events which it represents. A task can wait for an event by calling procedure *ajevwat*. Only tasks can wait for events. If a task wishes to wait for more than one event, all of the events must be contained in the one event group specified by the task in its call to *ajevwat*.

When the task calls *ajevwat* to wait for an event, it specifies the group id of the event group containing the events of interest. It provides a 16-bit mask identifying which of the events are of interest and a 16-bit value indicating the particular state of interest for each of the selected events. The task also specifies one of two types of match criterion to be used for event detection. Tasks can wait for any one of the selected events in the group to achieve its specified state. Alternatively, the task can insist that all of the selected events must exactly match their specified state before an event match can be declared.

Finally, the task calling *ajevwat* must also specify the interval, measured in system ticks, which it is prepared to wait for the event match to occur. Upon return from *ajevwat*, the task receives status indicating whether an event match occurred within the expected time interval. Note that a task will not be forced to wait if the state of the event flags meets the task's match criterion at the time of the call.

Events are signalled by tasks, ISPs and Timer Procedures. The event is signalled with a call to procedure *ajevsig*. The caller specifies the group id of the event group which contains the particular event. More than one event can be signalled in a single call to *ajevsig*. The caller specifies a 16-bit mask identifying the particular event flags in the group and a 16-bit value which specifies the new state of each of these selected event flags.

Whenever an event is signalled, the Event Manager determines if any tasks are waiting for any of the events in the particular group. If tasks are waiting on the group, the Event Manager checks the new state of the event flags to see if the event match criterion of any of the waiting tasks has been achieved. Whenever an event match is detected, the Event Manager wakes the task whose match criterion has been met. Providing that no tasks of higher priority are executing, the task will resume execution with an indication that the event combination for which it was waiting has occurred.

If a task needs to know the exact state of the event flags at the time its event match occurred or timed out, it can issue a call to the Event Manager procedure *ajevnt*.

A task, ISP or Timer Procedure can determine the current state of the event flags at any instant with a call to *ajevrd*.

If an event group is no longer required, the group can be deleted with a call to the Event Manager procedure *ajevdel*. The Event Manager will free the event group for reuse. The Event Manager will not allow you to delete an event group which has one or more tasks still waiting for events in the group.

You must be absolutely certain that no task, ISP or Timer Procedure is referencing the event group just as you go to delete it. Be aware that the event group id may immediately be reused by AMX for some other purpose.

7.3 Event Flag Application

The following example, coded in C, is provided to illustrate the use of the AMX Event Manager for event synchronization.

The example shows two tasks, A and B, which must be synchronized to the state of a motor. Task A must wait for the motor to be turned on. Task B must wait for the motor to be on and up to its maximum speed.

A 100 millisecond interval timer samples a motor control status register to determine the state of the motor. Whenever the motor state changes, the Timer Procedure signals the event. Note that the Timer Procedure signals both the on/off state and the motor speed simultaneously. It only signals changes in the motor state so that event synchronization overhead is minimized.

Note that the Timer Procedure receives its timer's id *timerid* and a parameter *unused*, neither of which is used by the procedure.

A Restart Procedure allocates an event group for motor control. Two of the event flags in the group are initialized to reflect the state of the motor at the time the system was started. The remaining event flags in the group are unused. A 100 millisecond periodic interval timer is created and started.

The creation and starting of Tasks A and B is outside the scope of this example. It is assumed that if the Restart Procedure is unable to create an event group for motor control, Tasks A and B will not be started. In our example, Tasks A and B assume that a valid event group id has been provided in variable *motorgroup*.

Note that bit 0 of the motor control status register determines if the motor is on or off. Bit 1 of the motor control status register determines if the motor is at maximum speed. For convenience, event flags 0 and 1 (bits 0 and 1 in the event group) are assigned to mirror the corresponding bits in the motor control status register.

```
#include "amx831cf.h"                  /* AMX C Interface Header      */

static AMXID motorgroup;
static unsigned char motorstatus;

#define MOTORPORT 0x65                 /* Motor status port           */
#define MOTORON 0x01                   /* Motor on                    */
#define MOTORMAX 0x02                  /* Motor at maximum speed      */

#define MOTOREVT (MOTORON + MOTORMAX)



void cdecl tpmotor(                    /* Motor Timer Procedure       */
AMXID timerid, void *unused)           /* Unused parameters           */
{
   unsigned int status;

   status = ajinb(MOTORPORT) & MOTOREVT;

/* If a change in motor status occurs                                 */
/*       Update motor status                                          */
/*       Signal that changes have occurred                            */

   if (status != motorstatus) {
         motorstatus = status;
         ajevsig(motorgroup, MOTOREVT, status);
         }
   }


void cdecl rrmotor(void)               /* Motor Restart Procedure     */
{
   AMXID timerid;

   motorstatus = ajinb(MOTORPORT) & MOTOREVT;

/* If an event group is available                                     */
/*       Set the initial event states to match the motor status       */
/*       If a 100 ms periodic timer can be created                    */
/*             Start it to go off at the next AMX system tick         */

   if (ajevcre(&motorgroup, motorstatus, "EVMT") == AEROK) {
         if (ajtmcre(&timerid, ajtmcnv(100), tpmotor,
                                       NULL, "TMMT") == AEROK)
               ajtmwr(timerid, 1);     /* Start timer                 */
         }
   }
```

```
void cdecl sttaskA(void)                /* Task A                       */
{
                                        /* Wait forever for motor on    */
   if (ajevwat(motorgroup, MOTORON, MOTORON, 0, 0) == AEROK) {
         :
         Motor is on.
         Process accordingly.
         :
         }
   }



void cdecl sttaskB(void)                /* Task B                       */
{
                                        /* Wait 5 seconds for motor     */
                                        /* on and at maximum speed      */
   if (ajevwat(motorgroup, MOTOREVT, MOTORON + MOTORMAX,
               1, ajtmcnv(5000)) == AERTMO) {
         :
         Motor not on and up to speed.
         Take recovery action.
         :
         }

   else {
         :
         Motor is on and up to speed.
         Process accordingly.
         :
         }
   }
```

This page left blank intentionally.

8. AMX Message Exchange Manager

8.1 Introduction

The AMX Message Exchange Manager provides a very flexible, general purpose mechanism for inter process communication and synchronization using prioritized messages. In particular, it offers an instant solution to a common problem frequently encountered in real-time applications: one or more processes (producers) having to asynchronously deliver requests of varying priorities for service to one or more servers (consumers) whose identity is unknown to the producer.

For example, assume that two printers are available to print reports and that requests for specific reports originate from a periodic Timer Procedure, from a data acquisition task and from a data base update task. The producers do not know which printer, if any, is free for use at the instant they must initiate their requests. Furthermore, if no printer is free, the data acquisition task and Timer Procedure are unable to wait for a printer to become available. The data acquisition task must be able to inject high priority print requests when errors are detected. How then to solve the problem?

The Message Exchange Manager readily provides the solution. A message exchange is created to act as a prioritized print request queue. The data acquisition task, Timer Procedure and data base task send their print requests to the message exchange. Two print server tasks, one for each printer, wait on the message exchange for the next print request. Each print server completes the generation of one report on its printer and then goes back to the exchange for the next request.

As just illustrated, the AMX Message Exchange Manager provides an extension to the inherent message passing services offered by AMX. The AMX kernel only allows a message to be sent to the particular task identified by the sender. The Message Exchange Manager extends the message delivery system to permit any task, ISP or Timer Procedure to receive a message. In so doing, it also removes the restriction requiring the message sender to identify the message receiver.

The Message Exchange Manager uses a message exchange to deliver messages. A message exchange consists of four message queues into which messages can be deposited. The message queues are ordered according to priority (0, 1, 2 or 3), message queue 0 being of highest priority.

Messages are delivered to the message queues in a message exchange in message envelopes. These are the same envelopes that are used by AMX to deliver messages to task mailboxes (see Chapter 3.9).

Any task, ISP, Timer Procedure or Restart Procedure can send a message to a message exchange. The sender indicates the priority of its message (0 to 3) thereby identifying the message queue into which it will be delivered.

Any task, ISP or Timer Procedure can request a message from a message exchange. Only tasks are allowed to wait for the arrival of a message if none is present in the message exchange when the task makes its request. A task can specify the priority at which it is willing to wait and the maximum time interval which it will wait for a message to arrive.

The task's wait priority is not to be confused with the message queue priority. The message queue priority determines the priority ordering of messages in the message exchange when no task is waiting for a message. The task's wait priority determines the order of tasks in the wait queue when more than one task is waiting for a message to arrive at an empty message exchange.

The flexibility of a message exchange comes from the fact that any number of consumers and producers can rendezvous at a message exchange without explicit knowledge of each other. Each consumer and producer only needs to know which message exchange to use. No consumer or producer owns the message exchange.

Constraints on the use of message exchanges are self-imposed by the system designer. You determine which producers and consumers will use each of your message exchanges. The only restriction imposed by the Message Exchange Manager is that only tasks are allowed to wait for a message to arrive at an empty message exchange.

The AMX Message Exchange Manager provides the following message exchange services:

ajmxcre	*AAMXCRE*	Create a message exchange
ajmxdel	*AAMXDEL*	Delete a message exchange
ajmxsnd *ajmxsndf* *ajmxsndp*	*AAMXSND*	Send a message to a message exchange
ajmxget	*AAMXGET*	Get a message from a message exchange (no wait)
ajmxwat	*AAMXWAT*	Wait for message to arrive at a message exchange (optional timeout)
ajmxtag	*AAMXTAG*	Find exchange id of a message exchange with a specific tag

Your use of the Message Exchange Manager is optional. If you intend to use it you must indicate so in your System Configuration Module. You must provide a hardware clock and include the AMX timing facilities.

Message exchanges can be predefined in your System Configuration Module which is processed by the Message Exchange Manager at startup. Message exchanges which are predefined are automatically created by the Message Exchange Manager. The message exchange id assigned to each predefined message exchange is stored in a variable which you must provide for that purpose.

8.2 Message Exchange Use

The Message Exchange Manager supports any number of message exchanges. The maximum number of message exchanges in a system is defined in your System Configuration Module (see Chapter 14.5). The defined maximum sets an upper limit on the number of actual message exchanges that can be created in your application.

A message exchange must be created by an application before it can be used. Restart Procedures, tasks, ISPs and Timer Procedures can create exchanges. It is recommended that only Restart Procedures and tasks be used to create message exchanges.

Create

Message Exchange Manager procedure *ajmxcre* is used to create a message exchange. The Message Exchange Manager allocates a message exchange and returns a message exchange id to the caller. The exchange id is a handle which uniquely identifies the message exchange. It is the responsibility of the application to keep track of the message exchange id for future reference to the exchange.

When a message exchange is created, you must specify the maximum number of message envelopes which are allowed to reside in each of the message exchange's four message queues. Message queue depths may range from 0 to 32767. If a particular queue is not used, set that queue's depth to zero. At least one message queue must have a non-zero depth.

Message queue depth has no effect on AMX memory requirements. That is, increasing a message queue depth does not increase memory needs. The message queue depths can be used to limit the number of messages allowed to be pending at the message exchange at any instant.

When a message exchange is created, you can provide a unique 4-character tag to identify the message exchange. The tag can be used subsequently in a call to *ajmxtag* to find the message exchange id allocated by the Message Exchange Manager to the particular message exchange.

When the Message Exchange Manager creates a message exchange, it sets all of the message exchange's message queues empty.

Send

A message is sent to a message exchange with a call to procedure *ajmxsnd* (or its variations *ajmxsndf* or *ajmxsndp*). The sender must specify the message priority (0 highest; 3 lowest) thereby indicating the message queue into which the message will be delivered. If the specified message queue is full or has been defined to have a depth of zero, the sender will be given an error indication.

A message is a set of parameter bytes located contiguously in memory. The maximum number of parameter bytes in a message is configured by you in your System Configuration Module (see Chapter 14.4). These parameter bytes are completely application dependent. The message size exactly matches that used for task mailbox messages (see Chapter 3.9).

Messages are sent to message exchanges in AMX message envelopes. The Message Exchange Manager gets a free message envelope from the common pool of envelopes maintained by AMX. You must therefore be sure to allocate enough message envelopes to meet the needs of all of your message exchange messages as well as your task messages.

Receive

Any task, ISP or Timer Procedure can get a message from a message exchange by calling procedure *ajmxget*. If the message exchange has any messages, the highest priority message (0 highest; 3 lowest) which arrived first will be given to the caller. If the message exchange is empty, the caller will immediately receive an error indication.

Tasks can also call procedure *ajmxwat* to get a message from a message exchange. Only tasks are allowed to call this procedure because the caller may have to wait for a message to arrive at the message exchange. If the message exchange has any messages, the highest priority message (0 highest; 3 lowest) which arrived first will be given to the caller. If the message exchange is empty, the caller will be forced to wait for a message to arrive.

The task specifies the interval, measured in system ticks, which it is willing to wait for a message to arrive. An interval of 0 indicates a willingness to wait forever. The task must also specify the priority at which it is prepared to wait, zero being the highest priority.

The Message Exchange Manager will add the task to the message exchange's wait queue at the priority the task said it was willing to wait. Thus a task can preempt other tasks already waiting in the queue at lower priority.

When a message finally arrives at the message exchange, the Message Exchange Manager will give it to the task, if any, at the head of the message exchange's wait queue. A task switch may occur immediately if the task being given the message is of higher task priority than the currently running task.

If no message arrives at the message exchange before the timeout interval specified by a task expires, the task will be removed from the message exchange wait queue and will resume execution with a timeout indication.

Delete

When a message exchange is no longer needed, it can be deleted with a call to *ajmxdel*. The Message Exchange Manager will reject the attempt to delete the message exchange if any task is waiting on the message exchange wait queue or if any messages remain in any of the exchange's message queues. When the Message Exchange Manager deletes the message exchange, it marks the message exchange as invalid such that any subsequent reference to the message exchange will be rejected.

You must be absolutely sure that no producer or consumer is referencing the message exchange just as you go to delete it. Be aware that the deleted message exchange id may be immediately reused by AMX for some other purpose.

8.3 Message Exchange Application

The following example, coded in C, is provided to illustrate the use of the AMX Message Exchange Manager.

This example illustrates a solution to the problem posed in the introduction (Chapter 8.1). Two message processing tasks, A and B, accept and service the messages from the data acquisition and data base tasks and the Timer Procedure. The processing tasks expect messages to arrive within 5 seconds.

A Restart Procedure creates the message exchange with message queues 0, 2 and 3, each of depth 10. In our example, the tasks and Timer Procedure assume a valid message exchange id has been provided in variable *msgexch*. The Restart Procedure also creates and starts the Timer Procedure.

The creation and starting of tasks A and B and the creation of the data acquisition and data base tasks is outside the scope of this example. It is assumed that if the Restart Procedure is unable to create the message exchange, the tasks will not be started.

Since we do not know how the data acquisition task and the data base update task were created, we will assume that their task ids are available in global variables as would be the case if the AMX Configuration Builder had been used to create the tasks.

The Timer Procedure triggers the data acquisition task once a second and sends a time-stamp message to the message exchange at the highest priority, 0.

The data acquisition task samples the data, triggers the data base update task and sends a data sample message to the message exchange at medium priority 2.

The data base update task processes the sampled data, updates the data base and sends a data update message to the message exchange at low priority 3.

Note that the two tasks, *sttaskA* and *sttaskB*, share a common re-entrant task body, *sttask*. In practice, procedure *sttaskB* could be eliminated and Task B could be created with *sttaskA* as the task start address. The example is coded to more clearly illustrate that two tasks actually exist.

```
#include "amx831cf.h                    /* AMX C Interface Header      */

#define UMS 12                          /* User message size           */

extern AMXID dactid;                    /* Data acquisition task id    */
extern AMXID dbtid;                     /* Data base update task id    */

static AMXID msgexch;                   /* Message exchange id         */

struct appmsg {                         /* Message structure           */
   short int msgtype;                   /* Application message type    */
   short int rsv;                       /* Reserved for alignment      */
   long     data;                       /* Data sample                 */
   void     *dbp;                       /* Pointer to data base        */
   };

union msgu {
   long maxmsg[UMS/sizeof(long)];       /* Maximum message size        */
   struct appmsg umsg;                  /* User message                */
   };


                                        /* Timer Procedure             */
void cdecl tmrproc(AMXID tmrid, void *unused)
{
   union msgu msgbuf;                   /* Message buffer              */

   :
   Construct message in msgbuf.umsg
   :
   ajmxsnd(msgexch, 0, &msgbuf);        /* Send message at priority 0  */

   ajtrig(dactid);                      /* Trigger data acquisition task*/
   }


void cdecl rrmsg(void)                  /* Restart Procedure           */
{
   AMXID    tmrid;                     /* Timer id                    */

   if (ajtmcre(&tmrid,                  /* Create periodic 1 sec timer */
                    ajtmcnv(1000),
                    (void (*)())tmrproc,
                    NULL, "TIMR") == AEROK)

            ajtmwr(tmrid, 1);           /* Start timer immediately     */

                                        /* Create message exchange     */
   ajmxcre(&msgexch, 10, 0, 10, 10, "MSGX");
   }
```

```
void cdecl sttask(void)                 /* Common task body            */
{
   union msgu msgbuf;                   /* Message buffer              */
   int      status;

                                        /* Wait at priority 0 for      */
                                        /* up to 5 sec for message     */
   status = ajmxwat(msgexch, &msgbuf, ajtmcnv(5000), 0);

   if (status == AEROK) {
            :
            Process the message in msgbuf.umsg
            :
            }

   else if (status == AERTMO) {
            :
            Process timeout - no message in 5 seconds
            :
            }

   else {
            Message retrieval failed
            Process some other error condition
            }
   }


void cdecl sttaskA(void)                /* Task A                      */
{
   for (;;) sttask();                   /* Do sttask forever           */
   }


void cdecl sttaskB(void)                /* Task B                      */
{
   for (;;) sttask();                   /* Do sttask forever           */
   }
```

```
void cdecl dactask(void)              /* Data acquisition task       */
{
   union msgu msgbuf;                 /* Message buffer              */

   :
   Perform data acquisition functions
   :
   if (no_error) {
            :
            Construct message in msgbuf.umsg
            :
                                      /* Send message at priority 3  */
            ajmxsnd(msgexch, 3, &msgbuf);
            :
            }

   else {                             /* Error has occurred!         */
            :
            Construct message in msgbuf.umsg
            :
                                      /* Send error message at       */
                                      /* priority 0                  */
            ajmxsnd(msgexch, 0, &msgbuf);
            }

   ajtrig(dbtid);                     /* Trigger data base task      */
   }


void cdecl dbasetask(void)            /* Data base update task       */
{
   union msgu msgbuf;                 /* Completion message          */

   :
   Perform data base update functions
   :
   :
   Construct message in msgbuf.umsg
   :
                                      /* Send completion message at  */
                                      /* priority 2                  */
   ajmxsnd(msgexch, 2, &msgbuf);
   }
```

9. AMX Buffer Manager

9.1 Introduction

The AMX Buffer Manager simplifies the management of memory buffers in a real-time system. It provides a general mechanism for the allocation and control of fixed size buffers.

The AMX Buffer Manager provides fast, efficient access to multiple pools of buffers, each buffer representing a fixed size block of memory. This form of memory management meets the requirements of most typical applications and is best suited for real-time use in which memory block availability must be predictable and in which the penalties for memory fragmentation cannot be tolerated.

You define a set of buffer pools, each pool containing a set of buffers of uniform size. Buffer Manager procedures are called to obtain a buffer from a particular pool and to release it back to the pool when it is no longer required.

When released, the buffer is automatically returned by the Buffer Manager to the pool to which the buffer belongs. Buffer ownership can be increased so that more than one task can simultaneously own a shared buffer. Special facilities are provided to assure that if a buffer is owned by more than one task, it is only returned to its pool when the slowest owner finally releases it.

Speed of execution is not dependent on the number of pools or buffers. The number of buffers in a pool is limited by the fact that all buffers of a particular pool must reside in one 64K segment. The number of pools which can be supported is limited only by memory availability.

The AMX Buffer Manager provides the following buffer management services:

ajbcre	*AABCRE*	Create a buffer pool
ajbdel	*AABDEL*	Delete a buffer pool
ajbget	*AABGET*	Get a buffer from a specific buffer pool
ajbfre	*AABFRE*	Release a buffer
ajbgsz	*AABGSZ*	Get size of buffer
ajbau	*AABAU*	Add to buffer use count
ajbia	*AABIA*	Initialize all currently defined buffer pools
ajbip	*AABIP*	Initialize one buffer pool
ajbtag	*AABTAG*	Find buffer pool id of a buffer pool with a specific tag

Your use of the Buffer Manager is optional. If you intend to use it you must indicate so in your System Configuration Module.

Buffer pools can be predefined in your System Configuration Module which is processed by the Buffer Manager at startup. Buffer pools which are predefined are automatically created by the Buffer Manager. The buffer pool id assigned to each predefined buffer pool is stored in a variable which you must provide for that purpose.

9.2 Buffer Pool Use

The Buffer Manager supports any number of pools of buffers. The maximum number of buffer pools in a system is defined in your System Configuration Module (see Chapter 14.5). The defined maximum sets an upper limit on the number of actual buffer pools that can be created in your application.

A pool of buffers consists of any number of buffers of a uniform size measured in bytes. Any even buffer size greater than or equal to eight bytes is allowed, up to the limits of a 64K memory segment. All of the buffers in a pool must reside in the same memory segment.

Create Buffer Pool

A buffer pool must be created by an application before it can be used. Restart Procedures, tasks, ISPs and Timer Procedures can create buffer pools. It is recommended that only Restart Procedures and tasks be used to create buffer pools.

Buffer Manager procedure *ajbcre* is used to create a buffer pool. The Buffer Manager allocates a buffer pool and returns a buffer pool id to the caller. The pool id is a handle which uniquely identifies the buffer pool. It is the responsibility of the application to keep track of the pool id for future reference to the buffer pool.

When a buffer pool is created, you must specify the following parameters: the number of buffers in the pool, the size of each buffer and a pointer to RAM storage for all of the buffers in the pool.

When a buffer pool is created, you can provide a unique 4-character tag to identify the buffer pool. The tag can be used subsequently in a call to *ajbtag* to find the buffer pool id allocated by the Buffer Manager to the particular buffer pool.

When the Buffer Manager creates a buffer pool, it subdivides the allocated RAM storage into the required number of buffers.

All buffers in the pool are linked together on a free list. This free list is internal to the Buffer Manager, hidden from view. Associated with (but not part of) each buffer is a use count, also internal to the Buffer Manager. The use count is set to zero for each buffer to show that it is not in use.

Get Buffer

Once a buffer pool has been created, you may call procedure *ajbget* to get a buffer from the pool. The Buffer Manager unlinks a buffer from the pool's free list, sets the associated buffer use count to one and returns a pointer to the first byte of the buffer. You may then store and retrieve any data in the buffer as desired.

If there are no free buffers available in the pool when a request to get a buffer from the pool is made, the Buffer Manager returns an error indication to the caller.

Free Buffer

When the buffer is no longer required, you can call procedure *ajbfre* to release the buffer. You indicate the buffer to be released by specifying the pointer to the first byte of the buffer, the same pointer that was received when you originally acquired the buffer. The Buffer Manager decrements the use count associated with the buffer by one. If the use count becomes zero, the buffer is linked to the free list of the pool to which it belongs.

Use Count

Once you own a buffer, you may call procedure *ajbau* to increase the use count. Recall that when a buffer is first obtained, the use count is set to one. If the use count is then increased by one, the buffer will have to be released twice before it becomes free. In certain applications such as described in Chapter 9.3, this feature is essential.

Size

You may also call procedure *ajbgsz* to obtain the size of the buffer. Because the buffer was obtained from a pool with buffers of known size, this call is usually unnecessary. However, in applications in which the current owner of a buffer did not actually acquire the buffer from the Buffer Manager, it is convenient to be able to determine the buffer size.

Delete

If at some point, there is no longer a need for any of the buffers in the buffer pool, the entire buffer pool can be deleted with a call to procedure *ajbdel*. It is your responsibility to assure that none of the buffers in the pool are still being used at the time you delete the pool. Once a buffer pool has been deleted, all of the RAM storage provided by you when the buffer pool was created is available for reuse by the application.

You must be absolutely certain that no task, ISP or Timer Procedure is referencing the buffer pool just as you go to delete it. Be aware that the deleted buffer pool id may immediately be reused by AMX for some other purpose.

9.3 Buffer Applications

Consider the following example. A process control system using AMX has a printer on which errors and status messages are to be logged. These messages are generated by several tasks as they perform their process control functions. These tasks must not wait for the printer because they have more important work to do. Furthermore, each message size may exceed the standard AMX message size.

A printer task receives the messages from the other tasks and outputs them to the printer. The problem is to get these messages from the process control tasks to the printer task.

The Buffer Manager provides an ideal solution. A pool of buffers is defined and dedicated to error logging. As each process control task detects an error, it calls the Buffer Manager to obtain a buffer, fills the buffer with an appropriate error message and makes an AMX *ajsend* call, sending the buffer pointer in an AMX message envelope to the task mailbox of a printer task.

The printer task automatically executes every time an AMX message arrives in its task mailbox. The AMX message is delivered to the task on its stack. The printer task extracts the buffer pointer from the AMX message, outputs the contents of the buffer to the printer, calls the Buffer Manager to release the buffer and ends. Because the process control tasks need never wait for the printer task or printer, they are always available to perform their intended functions.

Now consider the following addition to this example. Suppose it is desired to log messages on both a display and printer. There are several ways this could be done.

1. Two buffers could be obtained from the Buffer Manager. The same message would be copied into both buffers. One would be sent to a printer task mailbox, the other to a display task mailbox.

2. A single buffer could be obtained from the Buffer Manager. It would be filled with a message and sent to the display task mailbox. The display task would output the message to the display and send the buffer to the printer task mailbox. The printer task would output the message to the printer and release the buffer.

3. A single buffer could be obtained from the Buffer Manager. The message would be copied into the buffer. The Buffer Manager would be called to increase the buffer use count by one, for a total use count of two. The buffer pointer would then be sent to both the printer task mailbox and the display task mailbox. Both tasks would output the message to their output device and release the buffer. When the slowest of these two tasks released the buffer, the use count would be zero and the buffer would be free.

These solutions each have their own advantages and disadvantages. Method 1 requires twice as many buffers as the other methods and requires extra processor time to copy the message twice. Method 2 cannot display the message simultaneously on both the CRT and printer. Also, method 2 requires the display task to know about the printer task's mailbox. Method 3 is the most desirable.

Can method 3 be improved? Having the process control tasks know about the display and printer makes future system modification difficult. Instead, a single reentrant message routing procedure could be added to process all messages and determine if they should be sent to the display task mailbox or printer task mailbox or both. If the message routing procedure decided to send the buffer to both, it would call the Buffer Manager to increase the buffer use count before passing the buffer on. As in method 3, the display and printer tasks would only have to output the buffer, release it and then end. The process control tasks would only have to get a buffer from the Buffer Manager, fill it with a message and call the message routing procedure to send it to the appropriate task mailboxes.

What if the messages have to be sorted by priority? Just use the task mailbox priorities offered by the AMX message passing services.

9.4 Buffer Manager Caveats

Although the Buffer Manager attempts to check as many error conditions as possible, it cannot protect against a bad system design. However, if a little care is taken during system design, the Buffer Manager can help make a system more reliable than one that uses an ad hoc buffer management method.

The most important concept to understand about the Buffer Manager is **buffer ownership**. The Buffer Manager owns all free buffers. You must never modify these in any way.

Once you get a buffer (with a call to the Buffer Manager), you own the buffer. You may modify the contents of the buffer while it is owned. Once you release the buffer or pass it to another concurrently executing task or Interrupt Service Procedure, then the buffer is no longer owned by you and may not be modified or released or have its use count altered by you in any way.

If the use count is increased by the owner of a buffer, then several simultaneous owners are allowed. However, when there are several concurrently executing owners, the buffer content must not be modified unless your design is such that each owner can write to its own private portion of the buffer. The only permitted operations are reading the buffer and releasing it. Each owner should either release the buffer or pass the ownership on to another routine that will release it.

These restrictions are characteristic of message passing between concurrently executing routines in general, not just of the AMX Buffer Manager. To illustrate the necessity of these restrictions, a few examples of incorrect usage follow.

1. Task A gets a buffer, fills it, passes it to Task B and releases it. Task B attempts to use the data in the buffer.

   The problem is that Task A released the buffer after it passed ownership to Task B. Thus Task B will receive a buffer which it does not own. If Task B modifies the buffer, then the Buffer Manager may produce unpredictable results the next time it tries to use the buffer.

   The solution is to have Task B release the buffer when it is finished with it. Task B may do this because it owns the buffer after it receives it from Task A.

2. Task A gets a buffer, fills it, passes it to Task B, increments the use count by one and passes the buffer to task C. Tasks B and C use the buffer and release it.

   The problem is that Task A has passed ownership to Task B without first retaining the ownership by increasing the use count. It then increases the use count of a buffer it doesn't own. In the meantime, Task B may have finished using the buffer and released it.

   The solution is to have Task A expand the ownership by increasing the use count before passing the buffer to Task B. Task A then retains one half of the ownership and may pass this ownership on to Task C.

10. AMX Memory Manager

10.1 Introduction

The AMX Memory Manager simplifies the management of memory in an AMX system. It provides a general mechanism for the dynamic allocation and control of memory and is specifically designed for use in a multitasking environment.

Multitasking adds particular difficulties to the general problem of memory allocation. For example, high level language memory management procedures, such as C's *malloc*, *calloc* or *free*, cannot be called from concurrently executing tasks since the procedures are usually not reentrant.

The Memory Manager resolves these difficulties. The Memory Manager's general allocation procedures allow concurrently executing tasks to allocate and free blocks of memory.

The Memory Manager allocates memory from a memory pool containing a set of one or more sections of contiguous memory. Memory Manager procedures are called to obtain a memory block of any size from the memory pool and to release it back to the pool when it is no longer required.

When released, the memory is automatically returned by the Memory Manager to its memory pool. A memory block use count is maintained to allow a single block to have several owners and to prevent the block from being released until it is discarded by all owners. Hence, more than one task can simultaneously own a shared piece of memory. If a memory block is owned by more than one task, it is only returned to the memory pool when the slowest owner finally releases it.

The AMX Memory Manager maintains a single memory pool. The amount of memory in the pool is limited only by memory availability.

A particular unique feature of the Memory Manager permits any block of memory (including those acquired from the memory pool controlled by the Memory Manager) to be treated as a separate memory section from which smaller private blocks can be dynamically allocated. This feature allows a task which has well defined memory requirements to reserve a memory block for its own private use. The task can then call on the Memory Manager to control the allocation and release of smaller blocks from within the larger private block.

> Note
>
> The AMX 86 Memory Manager provides fast and efficient allocation of paragraph (16-byte) aligned blocks of memory for 80x86 systems with 16-bit memory addressing.
>
> When used with the VAutomation Turbo86 processors with 24-bit memory addressing, memory blocks are allocated with 16-byte alignment, *not* page (256 byte) alignment. For additional information, see the AMX 86 Tool Guide for Paradigm toolset *PX*.

The AMX Memory Manager provides the following memory management services:

ajmget	*AAMGET*	Get a block of memory
ajmfre	*AAMFRE*	Release a block of memory
ajmgsz	*AAMGSZ*	Get the size of a block of memory
ajmau	*AAMAU*	Add to block use count
ajmset *ajmsetf*	*AAMSET*	Set memory to a pattern
ajmhan	*AAMHAN*	Create a handle to a private memory section
ajmgeh	*AAMGEH*	Get a block of memory from a private memory section

Your use of the Memory Manager is optional. If you intend to use it you must indicate so in your System Configuration Module.

The memory for the memory pool controlled by the Memory Manager is assigned to the pool by the Memory Assignment Procedure which you identify in your System Configuration Module. The Memory Manager calls your procedure at startup to acquire all of the memory over which it will have control.

10.2 Nomenclature

The following nomenclature has been adopted by the Memory Manager.

A **Memory Section** is a contiguous, double word aligned area of Random Access Memory Random (RAM) which has been specified by the user to be under the control of the Memory Manager. A section can exceed 64K bytes in size.

The AMX **Memory Pool** is the private collection of one or more memory sections under the control of the Memory Manager from which tasks can request memory to be allocated for their private use.

A **Memory Block** is a double word aligned, contiguous portion of memory allocated by the Memory Manager from a memory section for use by a task. A memory block can exceed 64K bytes in size.

A **Header** (i.e. Memory Block Header) is an area of RAM associated with (but not part of) a memory block. It contains control information which is private to the Memory Manager and not accessible to any task.

A **Block Use Count** is an integer associated with (but not part of) a memory block. It is used by the Memory Manager to keep track of the number of owners of the memory block.

A **Memory Handle** is a 32-bit identifier provided by the Memory Manager to identify a memory block which is private to a task but still under the control of the Memory Manager.

10.3 Memory Allocation

The Memory Manager maintains a single memory pool. The memory pool consists of any number of memory sections of varying sizes measured in bytes. Any memory section size which is a multiple of 4 and greater than or equal to 64 bytes is allowed. Usually the memory pool consists of a single large memory section.

Memory sections are given to the Memory Manager by your Memory Assignment Procedure at startup. Once AMX is up and running, memory sections cannot be added or removed from the memory pool.

When the Memory Manager assigns a memory section to its memory pool, it treats the memory section as one large free memory block linked together with other free blocks on a free list. This free list is internal to the Memory Manager, hidden from view.

Get Memory

Once the Memory Manager's memory pool has been initialized, a task can call the Memory Manager procedure *ajmget* to get a block of memory of any size from the pool. Only tasks are allowed to call the Memory Manager to acquire memory blocks.

If the Memory Manager is able to find a block of memory in the memory pool large enough to meet the caller's requirements, the caller will be given a pointer to the memory block and an indication of the actual size of that block. The block may be marginally larger than the size requested. The memory block use count is set to one.

If the Memory Manager cannot locate a memory block in the memory pool large enough to meet the caller's requirement, it returns an error indication to the caller and identifies the largest block which is available in the pool at that instant. Note, however, that if the calling task immediately requests a block of memory equal to this largest memory block size, the request may still fail because, in a multitasking system, other higher priority tasks may have grabbed some of that memory before the task can make another request to the Memory Manager.

Free Memory

When use of a memory block is no longer required, the owner can call procedure *ajmfre* to release the block. The caller specifies the block to be released by passing the Memory Manager the same pointer that it received when the Memory Manager originally allocated the block.

The Memory Manager decrements the memory block's use count and, if the count goes to zero, returns the block to its memory pool. If there are any free memory blocks adjacent to the released block, they are merged with the block being released to form a larger block.

Once a block is released, no task in the system can make reference to it. The block enters the private domain of the Memory Manager.

Use Count

When the Memory Manager allocates a block of memory for the use of a task, it sets the block's use count to one. The block owner may call the Memory Manager procedure *ajmau* to increase the use count. If the use count is increased by one, the block will have to be released twice before it becomes free.

The block use count provides the key to memory block ownership. The Memory Manager owns all free blocks in its memory pool. One or more tasks can own an already allocated memory block. This concept of block ownership is the same as that of buffer ownership provided by the AMX Buffer Manager. Refer to Chapters 9.3 and 9.4 for examples of the proper use of this ownership concept.

Size

The Memory Manager procedure *ajmgsz* can be used to obtain the size of a particular memory block. This feature can be useful if one task is given ownership of a memory block by another task. The new block owner can check the size of the memory block to be sure that it meets its requirements. If a task corrupts the contents of any memory location outside the bounds of the memory block, the effects are unpredictable and potentially disastrous.

Once allocated, a memory block cannot grow or shrink in size. Hence, the size of a memory block cannot be dynamically adjusted by its owner.

10.4 Private Memory Allocation

A particularly unique feature of the Memory Manager permits any block of memory (including those acquired from the Memory Manager) to be treated as a memory section from which smaller private blocks can be dynamically allocated.

To use this feature, a task calls procedure *ajmhan* giving it a pointer to a private area of memory whose access is to be controlled by the Memory Manager. Blocks allocated by procedure *ajmget* or *ajmgeh* are suitable for this purpose. The caller must also specify the size of the memory area provided.

The Memory Manager takes the memory area which it has been given and converts it into a memory section for the private use of the task. This private section is identified by a memory handle, a 32-bit identifier which is returned to the caller. As long as the task does not make the memory handle public, the section remains private to the task.

The memory handle must be used to allocate smaller blocks from the private memory section. Memory Manager procedure *ajmgeh* is used to acquire a memory block from a private memory section identified with a handle. Once a private memory section has been created, any task having the handle can acquire a memory block from the private section. It is up to the task which created the private memory section to determine which tasks are given the handle.

10.5 Memory Assignment

The sections of memory which make up the memory pool controlled by the Memory Manager must be provided by your application. The Memory Manager makes no assumptions concerning the whereabouts of the memory over which it has control.

Sections of memory are assigned to the Memory Manager by your application. You must provide a Memory Assignment Procedure which will be called by the Memory Manager to acquire the memory sections over which it is to have control. A definition of this procedure is provided in Chapter 10.6.

If no Memory Assignment Procedure is provided, the Memory Manager will always report that no memory is available whenever any task requests a memory block.

When your AMX system is first started, the Memory Manager has no memory to allocate to tasks. When AMX is started, the Memory Manager determines if you have provided a Memory Assignment Procedure. If one was provided, the Memory Manager calls it to acquire a memory section. Your procedure must return a pointer to a memory section and indicate its size. The manner in which your procedure derives section pointers is up to you.

The Memory Manager initializes the section returned by your Memory Assignment Procedure. If the size of a section is less than the minimum required by the Memory Manager, the section is ignored. Your Memory Assignment Procedure is then called again. This process proceeds until your procedure returns an indication that no more sections are available.

Once your Memory Assignment Procedure indicates that there are no more sections available, the Memory Manager ceases its calls to your procedure. At that point, the Memory Manager has reached the upper limit of the memory over which it has control.

The Memory Manager will subsequently attempt to allocate a memory block for any calling task from the memory sections which it has acquired. If the first memory section has no free blocks large enough to meet the needs of the calling task, the next memory section is checked to see if it meets the requirements of the caller. This process is repeated until a block can be allocated to the caller or until all initialized sections have been exhausted.

Subsequent requests to the Memory Manager for a memory block will fail if a memory block of adequate size cannot be found in any of these initialized memory sections.

10.6 Memory Assignment Procedure

An application Memory Assignment Procedure must be provided to dynamically assign memory sections to the Memory Manager. The Memory Assignment Procedure is called by the Memory Manager when AMX is started, prior to execution of any of your Restart Procedures.

The name of your Memory Assignment Procedure must be provided in your System Configuration Module (see Chapter 14.5).

The Memory Assignment Procedure receives two pointers as parameters. The parameters point to storage for the size of the memory section and for a pointer to the section. The Memory Assignment Procedure must return the size of the memory section in bytes and a pointer to the memory section in the storage provided. The procedure must also return a non-zero integer if a valid memory section is provided. The minimum section size is 64 bytes which, in practical terms, provides no useable memory.

If the Memory Assignment Procedure cannot assign another memory section, it must return an integer result of zero without modifying the storage provided for the memory size.

Upon entry to the procedure, the pointers to storage for the memory size and memory section pointer are present on the stack. Upon entry to the procedure, the following conditions exist:

Interrupts reflect the state specified at the time AMX was launched. If AMX was launched with launch parameter *AMLPIE*, interrupts will be enabled. Otherwise interrupts will be disabled.

All registers are free for use.
DS,ES *DGROUP* Segment
SS:SP AMX Kernel Stack ready for use
BP Offset of parameters on stack
The direction flag is set to forward.

Your Memory Assignment Procedure can be coded as a Large or Medium model C procedure with formal parameters as follows.

```
int cdecl memproc(
long *memsizep,                 /* Pointer to memory size storage       */
char **memptrp)                 /* Pointer to memory pointer storage    */
{
   :
   :
   if (another memory section is available) {

            *memsizep = <size of memory section in bytes>;
            *memptrp = <pointer to memory section>;

            return(1);
            }

   return(0);                   /* No memory left                       */
   }
```

The Memory Assignment Procedure can be coded in assembler as a *FAR* procedure.

```
USER_CODE SEGMENT BYTE 'CODE'
;
          ASSUME   CS:USER_CODE
;
          PUBLIC   MEMPROC
;
PARAM     STRUC                    ;Define parameters on stack
          DW       ?               ;Save BP
          DD       ?               ;Return address
MEMSIZEP  DD       ?               ;A(storage for section size in bytes)
MEMPTRP   DD       ?               ;A(storage for memory section pointer)
PARAM     ENDS
;
MEMPROC   PROC     FAR
          PUSH     BP
          MOV      BP,SP           ;SS:BP = A(parameters on stack)
          :
          Set AX = 0 if memory section is not available
          :
          OR       AX,AX
          JZ       MEMEXIT         ;No section; AX = 0
          :
          Let DX:AX = size of memory section in bytes
          LES      BX,[BP].MEMSIZEP  ;ES:BX = A(memory size)
          MOV      WORD PTR ES:[BX],AX
          MOV      WORD PTR ES:[BX+2],DX
          :
          Let DS:SI = A(memory section)
          LES      BX,[BP].MEMPTRP   ;ES:BX = A(memory pointer)
          MOV      WORD PTR ES:[BX],SI
          MOV      WORD PTR ES:[BX+2],DS
          MOV      AX,1              ;AX <> 0
;
MEMEXIT:  POP      BP
          RET
;
MEMPROC   ENDP
;
USER_CODE ENDS
```

11. AMX Circular List Manager

11.1 Circular Lists

The AMX Circular List Manager provides a general circular list facility for use by application program modules.

Circular lists must be located in alterable memory (RAM).

A circular list is a data structure used by an application to maintain an ordered list of 8-bit, 16-bit or 32-bit elements. The elements are stored in slots in the list. Each list contains a fixed, user defined number of slots.

The AMX Circular List Manager provides the following list manipulation procedures:

ajrstl	*AARSTL*	Reset (initialize) a circular list
ajabl	*AAABL*	Add to bottom of circular list
ajatl	*AAATL*	Add to top of circular list
ajrbl	*AARBL*	Remove from bottom of circular list
ajrtl	*AARTL*	Remove from top of circular list

The Circular List Manager procedures are reentrant permitting them to be shared by concurrently executing tasks, ISPs and Timer Procedures.

Examples

Elements can be added to the top and removed from the bottom of a circular list. This makes these lists particularly attractive for character buffering by ISPs. Received characters can be added to the bottom of the circular list and removed from the top. If the process that removes a character decides that it cannot yet handle the character, it can return the character to the top of the circular list.

16-bit lists are useful for handling both a character and its receiver status. When a character is received, the character and its status are added to the bottom of a circular list. The task processing received characters pulls a 16-bit element from the top of the circular list to get each character and its status.

32-bit lists can be used to manage pointers to more complex structures. For instance, a sawmill application might use two lists to keep track of log measurements and resulting lumber counts in a Log Description Block, a private application structure. As logs are measured, their dimensions are inserted into a Log Description Block. A pointer to the block is added to the bottom of a circular list which serves as input to the cutting process. This process removes a pointer to the next available Log Description Block from the top of this list. Once the log is cut, the log's lumber results are added to the block and the block pointer is added to the bottom of another circular list for lumber tally processing.

11.2 Circular List Use

A circular list is created by an application with a call to procedure *ajrstl*. The caller must provide three parameters: the number of slots in the list, the size of each slot (1, 2 or 4 bytes) and a pointer to RAM storage for the circular list. The number of slots in a circular list is limited only by available memory and by the fact that the list must reside within one 64K byte segment in memory.

The RAM storage area must be 16-bit aligned and provide *(n*s)+8* bytes where *n* is the number of slots and *s* is the slot size (1, 2 or 4).

The Circular List Manager creates the list in the storage provided and sets the list empty. The Circular List Manager procedures can then be used to add and remove elements of the defined size at the top and/or bottom of the list.

The Circular List Manager procedures provide the caller with the status of the list following each call. When adding an element to the list, the caller will receive notice if the list is already full. If the element is added to the list, the caller will be notified if the list has just become full.

When removing an element from the list, the caller will receive notice if no element was on the list. If an element is retrieved from the list, the caller will be notified if the list has just gone empty.

11.3 Circular List Structure

Circular lists are application data structures which are only accessible by calls to the AMX Circular List Manager. The internal structure of the list is private to the Circular List Manager.

Lists can be created dynamically or statically by any application program.

The following examples illustrate 8-bit, 16-bit and 32-bit lists created in C. *NSLOT* is defined to be the number of slots in each list.

```
typedef char SLOT1;                     /* 1 byte slot              */
typedef short int SLOT2;                /* 2 byte slot              */
typedef long SLOT4;                     /* 4 byte slot              */

#define NSLOT 64                        /* 64 slots in the list     */


struct {                                /* List of 1 byte slots     */
   short int header[4];
   SLOT1    slots[NSLOT];
   } bytelist;

struct {                                /* List of 2 byte slots     */
   short int header[4];
   SLOT2    slots[NSLOT];
   } shortlist;

struct {                                /* List of 4 byte slots     */
   short int header[4];
   SLOT4    slots[NSLOT];
   } pntrlist;
```

Bytelist is a circular list of *NSLOT* 8-bit elements.
Shortlist is a circular list of *NSLOT* 16-bit elements.
Pntrlist is a circular list of *NSLOT* 32-bit elements.

The same lists can be coded in assembly language as follows:

```
USER_DATA SEGMENT WORD 'DATA'
;
; Circular Lists must be in program data
;
NSLOT       EQU      64
;
            EVEN                        ;Force word alignment
BYTELIST    LABEL    WORD
            DW       4 DUP(?)           ;Header
            DB       NSLOT DUP(?)       ;Slots
;
            EVEN                        ;Force word alignment
WORDLIST    LABEL    WORD
            DW       4 DUP(?)           ;Header
            DW       NSLOT DUP(?)       ;Slots
;
            EVEN                        ;Force word alignment
PNTRLIST    LABEL    WORD
            DW       4 DUP(?)           ;Header
            DD       NSLOT DUP(?)       ;Slots
;
USER_DATA   ENDS
```

12. AMX Linked List Manager

12.1 Introduction

The Linked List Manager provides a general set of fast linked list services suitable for use in real-time systems. The Linked List Manager removes the tedium and potential for serious error inherent in many applications in which list maintenance is implemented by each programmer in unique and varying ways.

The Linked List Manager offers a feature not often found in similar utilities. Objects manipulated by the Linked List Manager can concurrently reside on more than one list.

The Linked List Manager also resolves list manipulation races which frequently occur in real-time multitasking applications. The Linked List Manager assures that no list linkages will be corrupted because of races between tasks, ISPs or Timer Procedures attempting to manipulate the same list.

The Linked List Manager supports doubly linked lists in which all objects on a list are linked in forward and backward directions.

The Linked List Manager also supports keyed lists in which the order of objects in the list is determined by an ordering key provided by the application.

The AMX Linked List Manager provides the following list manipulation procedures. The procedures are reentrant permitting them to be shared by concurrently executing tasks, ISPs and Timer Procedures.

ajlcre	*AALCRE*	Create an empty list
ajlinsh	*AALINSH*	Insert at the head of a list
ajlinst	*AALINST*	Insert at the tail of a list
ajlinsc	*AALINSC*	Insert before the specified object on list
ajlrmvh	*AALRMVH*	Remove from the head of a list
ajlrmvt	*AALRMVT*	Remove from the tail of a list
ajlrmvc	*AALRMVC*	Remove specified object from a list
ajlhead	*AALHEAD*	Find current head of a list
ajltail	*AALTAIL*	Find current tail of a list
ajlnext	*AALNEXT*	Find next object on a list (walk towards tail)
ajlprev	*AALPREV*	Find previous object on a list (walk towards head)
ajlinsk	*AALINSK*	Insert into a keyed list
ajlordk	*AALORDK*	Reorder object in a keyed list
ajlmerg	*AALMERG*	Merge two lists

12.2 Linked Lists

Terminology

A **list header** is a structure provided by the application to be used to anchor a list. The list header is used to identify a list. The content of the list header is private to the Linked List Manager.

An **object** is an application data structure which represents the elements which reside on a list. For example, AMX maintains a list of Task Control Blocks (TCBs). Each TCB is an instance of a data structure representing the state of a task. Hence, the TCB qualifies as an object.

A **list node** is a structure embedded in an object. The list node is used to link the object into a simple doubly linked list. The content of the list node is private to the Linked List Manager.

A **key** is a 16-bit unsigned integer which is used to determine the order of objects in a keyed list. Each object in a keyed list must have a key. Objects in a keyed list are ordered such that key values are monotonically increasing from the head of the list to the tail of the list.

A **key node** is a structure embedded in an object. The key node is used to link the object into a keyed list. The object's key resides in the key node. The content of the key node is private to the Linked List Manager.

The **head** of a list is the first object on a list. The **tail** of a list is the last object on a list. An empty list has no head or tail. If only one object is on a list, it is both the head and tail of the list.

The **node offset** is the offset (i.e. displacement) into an object at which the list node (or key node) resides. The node offset assigned for a particular list is fixed. Any object which can reside on the list must have a list node (or key node) in the object at the node offset assigned to that list.

Figure 12.2-1 illustrates three doubly linked lists of apples and oranges. All apples and oranges reside on a fruit list. Fresh apples or oranges reside on a fresh list. Rotten apples or oranges reside on a rotten list.

The list objects are assumed to be data structures describing apples and oranges. Each object contains two list nodes: one for use with the fresh list or rotten list, the other for use with the fruit list.

Note that a single list node can be used for the fresh and rotten lists since these lists are mutually exclusive. All fruit list nodes reside at the same node offset in apple or orange objects. All fresh and rotten list nodes reside at the same node offset in apple and orange objects. The Linked List Manager lets you mix objects of different types on the same list as long as each object has a link node in it at the correct node offset for the particular list.

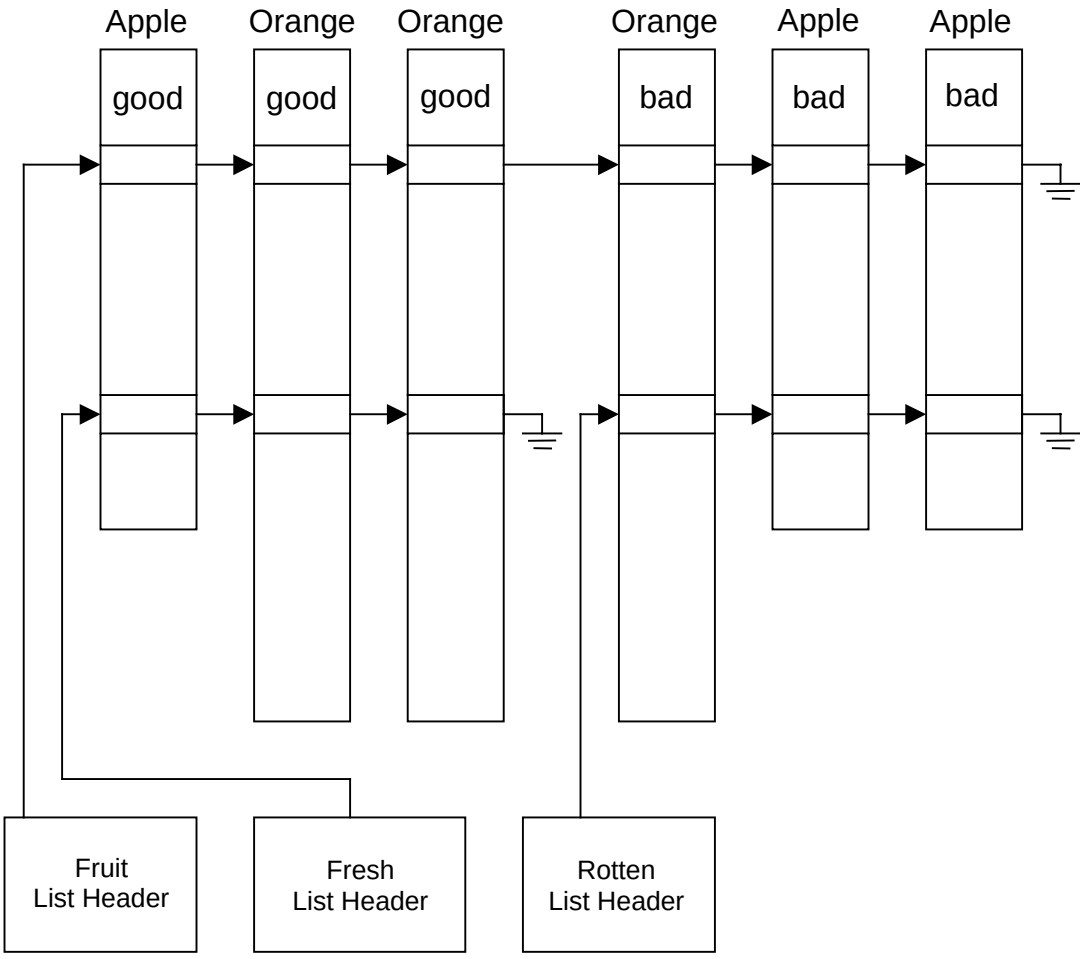


Figure 12.2-1 Doubly Linked Lists

12.3 Linked List Use

A list consists of a list header and objects linked to the list header by list nodes (or key nodes). Storage for the list header must be provided by you. A pointer to the list header acts as the list identifier.

An empty list is created by calling procedure *ajlcre* with a pointer to the list header storage. When the list is created, you must specify the node offset (byte displacement) at which link nodes (key nodes) will be found in objects which can reside on the list.

Once a list has been created, any object which has a link node (or key node) at that list's node offset can be added to the list.

Objects can be inserted at or removed from the head or tail of the list. You can always find the head or tail of the list. Given a pointer to any object on the list, you can find the next or previous object, insert another object ahead of it or remove it from the list.

If the objects have a key node at the list's node offset, then the list is a keyed list. All objects on a keyed list must have key nodes at the list's node offset. A new object is added to a keyed list according to the key provided in the call to *ajlinsk*. The object will be inserted into the list after all objects whose key is numerically less than or equal to the specified key. Thus keyed objects are added after all other objects with the same or lesser key.

The position of an object in a keyed list can be altered by calling *ajlordk* with a new key for the object. The list will be reordered moving the specified object to the correct position in the list according to its new key value. Reordering is usually faster than removing the object with *ajlrmvc* and re-inserting with *ajlinsk*.

The following example coded in C illustrates the use of the Linked List Manager. An object called *uobject* is defined with a key node at offset *keynode* in the object. An array of ten objects is provided. A keyed list *keylist* is created and the ten objects are added to the list in random order. The list is then perused to locate the actual position in the list of the sixth object in the array. The object is removed from the keyed list and added to the tail of a simple doubly linked list called *extlist*.

```
#include "amx831sd.h"                   /* AMX Structure Definitions   */

extern unsigned int random();           /* Random number generator     */


struct uobject {
   int      id;                         /* Object identifier           */
   int      data;                       /* Other application data      */
   struct amxlks keynode;               /* Key node                    */
   char     moredata[10];               /* More application data       */
   struct amxlns listnode;              /* List node                   */
   int      lastdata;                   /* Last application data       */
   };


/* Local variables                                                      */

static struct amxlhs keylist;           /* Keyed list header           */
static struct amxlhs extlist;           /* Extraction list             */

#define NUMOBJ 10                       /* Ten objects                 */

                                        /* Array of objects            */
static struct uobject objarray[NUMOBJ];
```

```
void example(void)
{
   int      i;
   struct uobject *objp;               /* Object pointer              */
   int      nodeofs;                   /* Node offset                 */


   nodeofs = (char *)(&objarray[0].keynode) - (char *)&objarray[0];

   ajlcre(&keylist, nodeofs);          /* Create empty keyed list     */


   nodeofs = (char *)(&objarray[0].listnode) - (char *)&objarray[0];

   ajlcre(&extlist, nodeofs);          /* Create empty extraction list*/


   objp = &objarray[0];                /* First object                */

   for (i = 1; i <= NUMOBJ; i++) {
            objp->id = i;              /* Insert object id            */
                                       /* Add to list in random order */
            ajlinsk(&keylist, objp++, random());
            }


   objp = ajlhead(&keylist);           /* Find head of list           */

   while (objp != NULL) {
            if (objp->id == 6)
                        break;         /* Found object of interest    */

            objp = ajlnext(&keylist, objp);
            }


   if (objp != NULL) {
            ajlrmvc(&keylist, objp);   /* Remove object from keylist  */
            ajlinst(&extlist, objp);   /* Add to extraction list      */
            }
   }
```

The following example coded in assembler illustrates the use of the Linked List Manager. The example mimics the operation of the previous C implementation. It illustrates the optimization effects which can be achieved by coding in assembler.

```
            INCLUDE  AMX831SD.DEF            ;AMX Structure Definitions
;
UOBJECT     STRUC
ID          DW       ?                       ;Object id
DATA        DD       ?                       ;Other application data
KEYNODE     DB       (SIZE AMXLKS) DUP(?)    ;Key node
MOREDATA    DB       10 DUP(?)               ;More application data
LISTNODE    DB       (SIZE AMXLNS) DUP(?)    ;List node
LASTDATA    DD       ?                       ;Last application data
UOBJECT     ENDS
;
OBJSIZ      EQU      SIZE UOBJECT            ;Size of an object
NUMOBJ      EQU      10                      ;Ten objects
;
USER_DATA   SEGMENT  WORD 'DATA'             ;Data segment
;
KEYLIST     DB       (SIZE AMXLHS) DUP(?)    ;Keyed list header
EXTLIST     DB       (SIZE AMXLHS) DUP(?)    ;Extraction list header
;
OBJARRAY    DB       (OBJSIZ*NUMOBJ) DUP(?)  ;Array of objects
;
USER_DATA   ENDS                             ;End of data segment
;
            PAGE
```

```
USER_CODE   SEGMENT BYTE 'CODE'                ;Code segment
;
            ASSUME   CS:USER_CODE, DS:USER_DATA, ES:USER_DATA
;
            EXTRN    RANDOM:FAR                ;Random number generator
;
EXAMPLE     PROC     FAR
            MOV      AX,SEG USER_DATA
            MOV      DS,AX                     ;Assure data access
            MOV      ES,AX
;
            MOV      CX,KEYNODE-(BYTE PTR ID)  ;Offset to key node
            MOV      SI,OFFSET KEYLIST         ;A(Keyed list)
            CALL     AALCRE                    ;Create keyed list
;
            MOV      CX,LISTNODE-(BYTE PTR ID) ;Offset to list node
            MOV      SI,OFFSET EXTLIST         ;A(Extraction list)
            CALL     AALCRE                    ;Create extraction list
;
            MOV      SI,OFFSET KEYLIST         ;A(keyed list)
            MOV      BX,OFFSET OBJARRAY        ;A(first object)
            MOV      DI,1                      ;i = 1

;
EXLP1:      MOV      [BX].ID,DI                ;Install object id
            CALL     RANDOM                    ;AX = random number
            MOV      CX,AX                     ;CX = key
;                                              ;ES:BX = A(object)
            CALL     AALINSK                   ;Insert in random order
            ADD      BX,OBJSIZ                 ;Next object
            INC      DI                        ;i++
            CMP      DI,NUMOBJ
            JLE      EXLP1                     ;More to go
            CALL     AALHEAD                   ;ES:BX = head of list

EXLP2:      MOV      AX,ES
            OR       AX,BX
            JZ       EXEND                     ;End of keyed list

            CMP      ES:[BX].ID,6
            JE       EXFND                     ;Found object of interest
            CALL     AALNEXT                   ;ES:BX = next object
            JMP      SHORT EXLP2
;
EXFND:      CALL     AALRMVC                   ;Remove object from list
            MOV      SI,OFFSET EXTLIST         ;A(Extraction list)
            CALL     AALINST                   ;Add to tail of list
EXEND:      RET
;
EXAMPLE     ENDP
;
USER_CODE   ENDS                               ;End of code segment
```

13. Advanced Topics

13.1 Fatal Exit

There are a number of conditions which, if encountered by AMX, are considered to be fatal. Any attempt by AMX to continue execution will lead to unpredictable results at best. All of these conditions cause AMX to force a branch to its fatal exit handler at *ajfatl*.

Insufficient Memory

The most common of these conditions occurs at startup. Many of the AMX managers include a Restart Procedure which is executed by AMX during its startup phase. Some of these managers must initialize a portion of the AMX Data Segment which is private to their needs. If any of these managers find that their needs exceed the size of the data segment provided in your AMX System Configuration Module, AMX takes its fatal exit. To proceed would imply using memory which does not belong to AMX.

Task Error Traps

AMX will take its fatal exit if a processor dependent task trap exception such as division by zero, arithmetic overflow or an array bound error occurs in any Restart Procedure, Interrupt Service Procedure or Timer Procedure.

If any of these errors occur in a task which does not have a task trap handler (see Chapter 4.5) for the corresponding error, AMX will take its fatal exit.

AMX and its managers avoid instructions which produce arithmetic overflow or array bound error exceptions. In the rare occasions when AMX or its managers use division, the division is guaranteed not to produce a divide fault.

Invalid Shutdown

If you started AMX requesting permanent execution (see Chapter 2.4) and then attempt to shut down your AMX system by calling *ajexit*, AMX will take its fatal exit path.

AMX Breakpoint Manager

If you include the AMX Breakpoint Manager in your system, it will attempt to install its handlers for the debug exception, breakpoint exception and possibly the NMI interrupt exception. If it is unable to install its handlers, the AMX Breakpoint Manager will force a fatal exit.

Application Faults

If your application encounters conditions which are deemed fatal, you can take the AMX fatal exit by calling procedure *ajfatl* or *AAFATL*. You can define your own negative fatal error codes to identify your fatal error conditions.

Fatal Exit Procedure

AMX allows you to provide a Fatal Exit Procedure of your own by specifying the name of your procedure in your System Configuration Module (see Chapter 14.5).

Whenever AMX or your application forces a fatal exit, AMX checks to see if you have provided a Fatal Exit Procedure. If one is present, AMX calls the procedure.

If no Fatal Exit Procedure exists or if the procedure returns to AMX, AMX will disable the external interrupt system and halt. Only a hardware reset can be used to recover.

The Fatal Exit Procedure must not make any use of AMX or any of its managers.

In general, there is little that the Fatal Exit Procedure can do. It certainly cannot rectify the situation. Any processing that it does should be done with the interrupts disabled if possible. If not, all interrupt sources should be reset or otherwise inhibited if hardware permits. Once all interrupt sources have been eliminated, the external interrupt system can be enabled. Only then is it acceptable to try restarting your AMX system.

Your Fatal Exit Procedure can be coded in assembly language as a *FAR* procedure. Upon entry to your Fatal Exit Procedure, the following conditions exist:

Interrupts are disabled.
All registers are free for use.

AX	Fatal error code (see *AERFXn* definitions)
DS,ES	*DGROUP* segment
SS:SP	Stack in effect at the time of the fatal exit

The direction flag is set to forward.

```
USER_CODE   SEGMENT BYTE 'CODE'
;
            ASSUME   CS:USER_CODE
;
            PUBLIC   FATALEX
;
FATALEX     PROC     FAR
            :
            Interpret fatal exit code in AX
            :
            RET                          ;AMX will halt at AMHALT
;
FATALEX     ENDP
;
USER_CODE   ENDS
```

Your Fatal Exit Procedure can be coded as a Large model C procedure as illustrated in the following example.

```
#include "amx831ec.h"               /* AMX Error Code Definitions     */

void cdecl fatalexit(
int error)                          /* Fatal exit error code          */
{
   :
   Inhibit all interrupt sources
   Provide an external indication of the fatal condition
   :
   }
```

Halt Locations

If AMX takes its fatal exit and you have not provided a Fatal Exit Procedure (or if your procedure returns to AMX), AMX will disable interrupts and halt at entry point *AMHALT*.

Entry point *AMHALT*, although private to AMX, is made *PUBLIC* to assist you in your system development and testing. This one entry point is the only location within AMX where a programmed halt can occur.

13.2 User Error Procedure

Most AMX procedures return error status to the caller. The error status is a signed integer.

AERER0K	= 0	No error
AERxxx	> 0	Warning: possible fault
AERxxx	< 0	Error: may be unrecoverable

The defined error codes are summarized in Appendix B.

AMX allows you to provide a User Error Procedure to trap all errors generated by calls to AMX functions. You do so by specifying the name of your procedure in your System Configuration Module (see Chapter 14.5).

Before returning an error or warning condition to the caller, AMX calls your User Error Procedure, if one has been provided. The User Error Procedure receives the AMX error code and the task id of the task executing at the time of the error. The task id will be *0* if the error occurred in an ISP call to AMX.

Upon entry to your User Error Procedure, interrupts are disabled to prohibit device interrupt, clock ISP and higher priority task activity until you have had an opportunity to examine the error condition. Your procedure should not enable interrupts without first disabling or resetting all interrupt sources. In all but the simplest of systems this may be impossible to do.

The User Error Procedure executes in the context of the task, ISP, Timer Procedure, Restart Procedure or Exit Procedure which made the errant AMX call. The User Error Procedure can only make AMX calls which an ISP is allowed to make.

In general, there is little that the User Error Procedure can do. It can, however, be extremely useful for locating faults in your application during initial testing. It is also useful for locating infrequently occurring error conditions which are not being checked by your code and hence are going undetected in an otherwise working system.

Your User Error Procedure can be coded as a Large model C procedure with formal parameters as illustrated in the following example.

```
#include "amx831cf.h"              /* AMX C Interface Header         */

void cdecl userror(
int error,                         /* AMX error code AERxxxx         */
AMXID taskid)                      /* Task id of current task        */
                                   /* (0 if error occurred in ISP)   */
{
   :
   Interpret the error code
   Interpret the task id
   :
   }
```

Your User Error Procedure can be coded in assembly language as a *FAR* procedure. Upon entry to your User Error Procedure, the following conditions exist:

Interrupts are disabled.
All registers are free for use.

AX	AMX error code (see *AERxxx* definitions)
DX	Task id of current task (0 if error detected in ISP)
DS,ES	*GROUP* segment
SS:SP	Stack in effect at the time of the error

The direction flag is set to forward.

```
USER_CODE   SEGMENT BYTE 'CODE'
;
            ASSUME   CS:USER_CODE
;
            PUBLIC   USERROR
;
USERROR     PROC     FAR
            :
            Interpret error code in AX
            Interpret task id in DX
            :
            RET                          ;Resume
;
USERROR     ENDP
;
USER_CODE   ENDS
```

> Note
>
> Warnings and timeouts are NOT trapped to your User Error Procedure. Appendix B identifies all *AERxxxx* codes which are not trapped.

13.3 Task Scheduling Hooks

AMX does not provide direct support for specific hardware extensions such as a math coprocessor or a memory management unit. Instead, AMX allows a set of application procedures to be connected to its Task Scheduler. These procedures can save and restore hardware dependent parameters specific to your application whenever a task switch occurs.

There are four critical points within the AMX Task Scheduler. These critical points occur when:

a task is started
a task ends
a task is suspended
a task is allowed to resume.

AMX allows a unique application procedure to be provided for each of these critical points. Pointers to your procedures are installed with a call to procedure *ajhook*. You must provide a separate procedure for each of the four critical points. Since these procedures execute as part of the AMX Task Scheduler, their operation is critical. These procedures must be coded in assembler using techniques designed to assure that they execute as fast as possible.

The AMX Task Scheduler calls each of your procedures with the same calling conventions.. Your procedures must be coded as *FAR* procedures.

Upon entry to your scheduling procedures, the following conditions exist:

Interrupts are disabled and must remain so.

DS:SI	A(Task Control Block)
SS:SP	Task stack free for use
AX,BX,CX,DX,ES	Free for use

All other registers must be preserved.

Register pair *DS:SI* gives you a pointer to the Task Control Block of the task which is being started, ended, suspended or resumed. Your procedures are free to use the task's stack. However, you must not attempt to use this stack to save information. For instance, popping the return address, pushing parameters onto the stack and then returning to AMX is not allowed.

If you have to save information as part of the task's state, you should use the storage in the Task Control Block reserved for the private use of your application (see Chapter 3.12). If necessary, provide an extension to your Task Control Block and install a pointer to the extension in the portion of the Task Control Block reserved for your use.

Once your procedures are installed, you will observe a degradation in the AMX task switching performance. Each call to your procedure will add the setup and calling overhead plus the time it takes your procedure to execute.

13.4 Abnormal Task Termination

A task is a procedure which is called by the AMX Task Scheduler. The task ends execution normally by returning to AMX. AMX provides procedure *ajend* which can be used by a task to end execution and return to AMX under circumstances in which its stack is deeply nested.

These two methods by which a task may end execution are considered normal termination. A task has to be running to end in this fashion. No other task can force a task, other than itself, to end.

AMX also provides a set of services which permit one task to force the abnormal termination of another task. These services are not to be treated lightly. Their description has been deferred to this chapter to indicate that they should be rarely invoked.

It is assumed that you have read Chapter 3 of this manual and are familiar with the AMX Task State Diagram presented in Figure 3.2-1.

The ability to arbitrarily terminate a task is one of the most abused privileges afforded by some multitasking systems. The AMX philosophy is to encourage well structured task designs in which there is rarely the need for an uncontrolled task termination. In many cases, a modification to your system design can eliminate the apparent need to arbitrarily terminate a task. However, occasionally system design does demand that an erring task be eliminated for the good of all others. The availability of a controlled task termination can greatly enhance your options in this case.

AMX offers the following task termination services:

ajtkstp	*AATKSTP*	Force a task to stop
ajtkill	*AATKILL*	Kill a task by forcing it to stop; flush its message mailboxes
ajtkdel	*AATKDEL*	Delete a task
ajtktrm	*AATKTRM*	Enable/disable abnormal task termination

The AMX termination services will not be included in your AMX system if your tasks do not call any of these procedures.

Stop a Task

A task can be stopped. The effect is the same as if the task had just issued a call to *ajend* to end its own operation. Only a task which is running, waiting or ready to execute can be stopped. A task can stop itself.

A task that has been stopped does not cease to exist. It simply ends operation. If it has outstanding trigger requests for execution, the task will be allowed to run again.

You must not stop any task which is waiting for a semaphore, an event group or a message exchange.

Kill a Task

A task can be killed. The task is first stopped as just described. All outstanding requests to the task for its execution are purged. The effect is the same as if the task continued to make calls to *ajend* to end its operation until finally there were no task execution requests remaining. A task can kill itself.

A task that has been killed does not cease to exist. Any new requests to trigger the task or to send messages to it will be honored.

You must not kill any task which is waiting for a semaphore, an event group or a message exchange.

Delete a Task

A task can be deleted. A task which is deleted ceases to exist. Its task id becomes invalid and may be assigned by AMX to some other newly created task. A task can delete itself.

You must not delete any task which is waiting for a semaphore, an event group or a message exchange.

Task Termination Procedure

To safeguard against abuses of its task termination services, AMX inhibits any task from being stopped, killed or deleted until the task itself indicates its willingness to be abnormally terminated. A task does this by calling procedure *ajtktrm* to inform AMX that it is prepared to be abnormally terminated.

When a task calls *ajtktrm* to allow its abnormal termination, it gives AMX a pointer to a Task Termination Procedure to be called by AMX whenever the task is stopped, killed or deleted. A task can subsequently inhibit its own abnormal termination by calling *ajtktrm* with a *NULL* procedure pointer.

Once a task enables abnormal termination, it remains enabled even if the task ends normally. The next time the task executes, it will still be able to be abnormally terminated. The previously defined Task Termination Procedure remains in effect until it is cancelled by calling *ajtktrm* to install a *NULL* pointer.

If a task is stopped or killed, AMX resets its Task Termination Procedure pointer to *NULL* thereby inhibiting any further requests to stop or kill the task until the task has had a chance to execute again and specify a new Task Termination Procedure.

A Task Termination Procedure can be coded as a Large or Medium model C procedure as illustrated in the following example. The procedure receives an integer reason code indicating whether the task is being stopped, killed or deleted. The mnemonics for these reason codes are provided in the AMX header file *AMX831SD.H*.

Upon entry to the Task Termination Procedure, the following conditions exist:

Interrupts are enabled.
All registers are free for use.
A reason code is provided as a parameter.
(see *ajtktrm* description in Chapter 16)
The task's stack in effect at the time of the termination request is in use.

The Task Termination Procedure must return to AMX. It must not call procedure *ajend* to end task execution.

```
#include "amx831sd.h"                  /* AMX Definitions             */

void cdecl termproc(                   /* Task Termination Procedure  */
int reason)                            /* Termination reason          */
{
   switch(reason) {

   case     AMCFE:
            :
            Stop task is occurring
            :
            break;

   case     AMCFK:
            :
            Kill task is occurring
            :
            break;

   case     AMCFD:
            :
            Delete task is occurring
            :
            break;

            } /* end of switch         */
   }
```

A Task Termination Procedure can be coded in assembler as a *FAR* procedure as indicated in the following example. The procedure receives an integer reason code indicating whether the task is being stopped, killed or deleted. The mnemonics for these reason codes are provided in the AMX header file *AMX831SD.DEF*.

Upon entry to the procedure, the following conditions exist:

Interrupts are enabled.
All registers are free for use.

AX	Reason code indicating stop, kill or delete task *(AMCFx)*
DS,ES	*DGROUP* segment
SS:SP	Task stack in effect at the time of the termination request

The direction flag is set to forward.

```
            INCLUDE  AMX831SD.DEF      ;AMX Definitions
;
;
USER_CODE SEGMENT BYTE 'CODE'
;
            ASSUME   CS:USER_CODE
;
TERMPROC    PROC     FAR               ;Task Termination Procedure
            :
            AX = AMCFE if stop task is occurring
            AX = AMCFK if kill task is occurring
            AX = AMCFD if delete task is occurring
            :
            RET
;
TERMPROC    ENDP
;
USER_CODE   ENDS
```

Termination Processing

AMX will only stop or kill a task which is running, waiting or ready to execute. A task can be deleted if it is in any of these states or idle. Occasionally, a request to terminate a task will occur while that task is performing some operation which AMX deems to be critical. When this occurs, AMX allows the task to complete the critical operation and then forces the abnormal termination. AMX does this so that none of its private resources, such as message envelopes, are corrupted or lost.

When a task is stopped or killed, AMX ensures that a task waiting for its message to be processed by the task being terminated, resumes and is not left blocked forever.

When a task is killed, AMX flushes each of the task's mailboxes. When doing so, AMX examines the message. If the message indicates that a task is waiting for the message to be processed, AMX allows the task to resume.

Deleting a task is a complex operation. AMX must make certain that the task is not deleted until all currently active transactions involving that task have been completed. For example, high priority Task A may be deleting medium priority Task B just as low priority Task C was interrupted in its attempt to trigger Task B. The task trigger must be allowed to complete in order that AMX task status not be compromised.

AMX meets these stringent requirements by requiring that you specify the priority at which a task is to be deleted. The task deletion priority must be below that of all other tasks with which it interacts but above any compute bound tasks which you may have. AMX drops the task to its deletion priority which forces all other tasks with which the task can interact to complete any outstanding operations before the deletion takes place. AMX then deletes the task.

Once a request to delete a task has been made, no new transactions involving the deleted task can occur. Any attempt to reference a task which has been marked for deletion will produce an *AERNST* error indication since the deleted task no longer exists as far as you are concerned. The actual task deletion may be delayed by AMX until current transactions involving the task can be completed.

Termination Warning

When AMX stops, kills or deletes a task, it does not release any of the system resources which that task may have within its control. Any interval timers, semaphores, event groups, message exchanges, buffers or memory blocks which the task has reserved must be released by the task's Task Termination Procedure when it is called by AMX.

13.5 Task Suspend/Resume

Some operating systems permit a task to suspend any task, including itself. This feature is then used to implement the equivalent of the AMX task wait procedure *ajwait*.

The ability to arbitrarily suspend a task is one of the most abused privileges afforded by other operating systems. The AMX philosophy is to encourage well structured task designs using the wide range of task synchronization facilities offered by AMX in which there is rarely the need for an uncontrolled task suspension. In most cases, a modification to your system design can eliminate the apparent need to arbitrarily suspend a task.

To facilitate porting system designs from other operating systems to AMX, the procedures *ajsusp* and *ajresum* are provided.

Any Restart Procedure, task, Timer Procedure or Exit Procedure can suspend any task in the system except the AMX Kernel Task. ISPs cannot suspend tasks. The task remains suspended until some task, ISP or Timer Procedure calls *ajresum* to force resumption of that task.

If a task is blocked (waiting for a task trigger or AMX message or in any AMX wait state) at the time it is suspended, the task will remain blocked (suspended) until an *ajresum* call attempts to let the task resume. If, while the task was suspended, the blocking conditions were removed, the task will resume (start) at the point at which it was blocked.

> Note
>
> Do not use *ajsusp* and *ajresum* for synchronization.
>
> For task synchronization, see Chapter 3.6.
> For ISP synchronization, see Chapter 4.4.

13.6 Breakpoint Manager

The AMX Breakpoint Manager can be used to improve the operation of the debugger that you use to test your AMX system. Your use of the Breakpoint Manager is optional; use it only if it augments your debugger as described below.

If you enjoy the use of an in-circuit emulator or hardware assisted debugger, you will not need to use the Breakpoint Manager. When breakpoints are encountered, operation of the target processor is suspended and you get a true snapshot of life in your AMX system at the time of the breakpoint. Since the processor operation is suspended, no further processing by the system under test is possible.

If you are using a resident debugger such as Microsoft CodeView or Borland Turbo Debugger operating on PC AT compatible hardware, then you should use the Breakpoint Manager to assist the debugger. Debuggers like these use software breakpoints and trace traps to provide their services. In the debugger's attempt to remain reasonably passive with respect to the underlying hardware, it leaves itself wide open to difficulties in testing interrupt driven pre-emptive multitasking applications.

When a breakpoint is encountered, the debugger may not inhibit clock interrupts because it prefers not to be so intrusive. The implication is that all clock activity continues to occur while you are stopped at a breakpoint. Consequently, in your AMX system, any task of higher priority than the one in which the breakpoint occurred may preempt the debugger and start (or resume) execution. This same effect will be noticed for all interrupt driven activity involving higher priority tasks.

The Breakpoint Manager resolves this problem. When a breakpoint or debug exception occurs, the Breakpoint Manager halts all AMX tasks and inhibits AMX clock service while you are in the debugger.

Unfortunately, the Breakpoint Manager has no easy means of determining when you have proceeded from a breakpoint. It monitors task switches and clock interrupts to detect an exit from the debugger. Once the exit has been detected, it removes all AMX tasks from the halt state and restores AMX clock service.

Using the Breakpoint Manager

The Breakpoint Manager is included in your application only if it is enabled in your User Parameter File. Use the AMX Configuration Builder to edit your User Parameter File (see Chapter 14.13). Enable the breakpoint support option in the Breakpoint parameter window and generate a new System Configuration Module. Then assemble the module and link your AMX system as described in the AMX 86 Tool Guide.

When you are finished testing your AMX system, you can remove the Breakpoint Manager. The simplest method is to relink your AMX system including object module *AA831BKE.OBJ* in your link specification prior to the AMX Library *AMX831.LIB*. This object module contains procedure stubs which satisfy the link requirements without forcing the Breakpoint Manager to be loaded from the library.

Alternatively, you can alter your System Configuration Module to exclude breakpoint support. Use the AMX Configuration Builder to edit your User Parameter File. Disable the breakpoint support option in the Breakpoint parameter window and generate a new System Configuration Module. Then assemble the module and relink your AMX system.

Breakpoint Exit Detection

The Breakpoint Manager monitors the breakpointed task's state to detect when an exit from the debugger has occurred. Whenever the task ends, waits or attempts to ready some other task, the Breakpoint Manager concludes that you have left the debugger and allows your AMX system to resume execution.

If your debugger inhibits clock interrupts at breakpoints, you can improve the Breakpoint Manager's exit detection as follows. Use the AMX Configuration Builder to edit your User Parameter File. Set the breakpoint entry delay to *-1* and the breakpoint exit delay to *1* in the Breakpoint parameter window. Generate a new System Configuration Module, assemble the module and relink your system. The Breakpoint Manager will assume that you have left the breakpoint as soon as the next AMX clock tick is detected.

Interrupt Masking

The Breakpoint Manager must be tailored to inhibit interrupts at breakpoints. Which interrupts are to be inhibited and how this is done is both system and hardware dependent.

When you include the Breakpoint Manager in your AMX system, module *AA831BKA.OBJ* is included from the AMX Library. This module contains procedure *AMBKOF* to inhibit selective interrupts at a breakpoint and procedure *AMBKON* to restore these interrupts upon resumption from the breakpoint.

Module *AA831BKA.OBJ* supports the use of AMX on PC/AT compatible hardware in which two Intel 8259 interrupt controllers are used to mask interrupts.

The 8259 interrupt disable masks *AMBKM1* and *AMBKM2* identify the interrupt request lines (IRQs) which will be inhibited at breakpoints. Bits 0 to 7 of mask *AMBKM1*, if set to 1, inhibit interrupts on IRQ0 to IRQ7 respectively. Bits 0 to 7 of mask *AMBKM2*, if set to 1, inhibit interrupts on IRQ8 to IRQ15 respectively on the slave 8259. By default, mask *AMBKM1* is set to *0FCH* and mask *AMBKM2* is set to *0FFH* to inhibit all interrupts except the clock on IRQ0 and the keyboard on IRQ1.

You must leave the keyboard interrupt enabled (*AMBKM1* mask bit 1 = 0) to permit PC debuggers to use the keyboard. You must leave the clock interrupt enabled (*AMBKM1* mask bit = 0) to permit proper resumption from breakpoints. You may wish to leave the interrupt enabled for the diskette (*AMBKM1* mask bit 6 = 0) or the hard disk (*AMBKM2* mask bit 6 = 0) to allow the debugger to access disk files.

If you are testing an AMX system on custom hardware, you may have to edit procedures *AMBKOF* and *AMBKON* in module *AA831BKA.ASM* to meet the needs of your hardware. Assemble the module and rebuild the AMX 86 Library to include your version of this module. Follow the instructions provided in Appendix A of the AMX 86 Tool Guide.

NMI Breakpoints

Some hardware debuggers may use the non-maskable interrupt (NMI) to generate breakpoints. The Breakpoint Manager can be tailored to support such debuggers by enabling its NMI handler.

When an NMI interrupt is detected, the Breakpoint Manager calls the NMI handler to determine if the interrupt is actually a breakpoint. In some 8086 PC configurations, the NMI interrupt is multiplexed to detect memory parity errors and to service a numeric coprocessor such as the Intel 8087, 80287 or 80387.

The NMI handler must determine the actual source of the NMI interrupt. If the handler determines that the interrupt is a breakpoint, the Breakpoint Manager halts all AMX task activity and then passes control to your NMI debugger.

If the NMI handler determines that the interrupt is not a breakpoint, the Breakpoint Manager allows your AMX system to continue execution and passes control on to the real NMI Interrupt Service Procedure.

The version of module *AA831BKA.OBJ* which is delivered with AMX assumes that the NMI interrupt is not used for debugging. If you are using the NMI interrupt for this purpose, you will have to edit file *AA831BKA.ASM* to enable NMI handling. The file contains a simple NMI handler suitable for use on PC/AT compatible hardware. If you intend to use an NMI debugger, you will have to edit flag *AMBKNM* in the file to be non zero. The file contains instructions defining how the editing should be done. Assemble the module and rebuild the AMX 86 Library to include your version of this module. Follow the instructions provided in Appendix A of the AMX 86 Tool Guide.

Compatible Debuggers

The Breakpoint Manager will operate with any debugger which exhibits the following characteristics.

The debugger must use breakpoint software interrupt (INT 3) or the NMI hardware interrupt (INT 2) to set breakpoints.

When a breakpoint is encountered, the debugger must switch to its own private internal stack which is different from all AMX task stacks.

The debugger must leave the interrupt controller (or equivalent) interrupt enable mask unaltered when it proceeds from a breakpoint.

14. AMX System Configuration

14.1 System Configuration Module

The AMX System Configuration Module defines the characteristics of your AMX system. The AMX Configuration Builder, described in Chapter 14.2, will create this module for you.

The System Configuration Module includes the following components.

The **User Parameter Table** provides AMX with the following information pertinent to the system you are creating.

Maximum number of tasks
Maximum number of timers
Maximum number of semaphores
Maximum number of event groups
Maximum number of message exchanges
Maximum number of buffer pools
Number of message envelopes
Message size
Kernel and Interrupt Stack sizes
Clock frequency
Time/Date Scheduling Procedure
Memory Assignment Procedure
User Error Procedure
Fatal Error Procedure

The **Restart Procedure List** defines all of your Restart Procedures and their order of execution. The **Exit Procedure List** defines all of your Exit Procedures and their order of execution.

If you wish, you can predefine one or more of any of the following components of your system. The object definition and memory storage, if any, required by AMX for the component are provided in the System Configuration Module.

Tasks (including tasks with mailboxes)
Timers
Semaphores (resource or counting)
Event groups
Message exchanges
Buffer pools

If you predefine any of these components, AMX will automatically create them when your system starts. The id assigned to each of the predefined components will be stored in unique **id variables** which are in the System Configuration Module.

If you predefine any tasks, the **task stacks** for these tasks will be in the System Configuration Module.

If you predefine any buffer pools, the **buffers** for these buffer pools will be in the System Configuration Module.

14.2 System Configuration Builder

The AMX Configuration Builder is a software generation tool which can be used to create your AMX System Configuration Module. The Builder helps to reduce total system implementation time by eliminating the manual generation process by which your System Configuration Module would otherwise have to be produced. The Builder consists of two components: the Configuration Manager and the Configuration Generator. The Configuration Manager is an interactive utility which allows you to define and edit all of the parameters which must be included in your System Configuration Module to use AMX and its Managers.

The configuration process is illustrated in the block diagram of Figure 14.2-1.

The Configuration Manager lets you define your system to meet your needs. It produces a text file called a User Parameter File. This file contains a cryptic representation of your description of your system.

The Configuration Manager is also able to read a User Parameter File allowing you to use the Configuration Manager to edit the content of a previously defined system description.

For convenience, the Configuration Manager has the ability to directly invoke its own copy of the Configuration Generator. The Configuration Generator reads your User Parameter File and uses the information in it to produce a source file called the System Configuration Module and a text file called the System Documentation Module.

The Configuration Generator uses a file called the System Configuration Template as a model for your System Configuration Module. This template file is merged with the information in your User Parameter File to produce your System Configuration Module.

The Configuration Generator also merges a file called the System Documentation Template with the information in your User Parameter File to produce your System Documentation Module, a text file summarizing the characteristics of your AMX configuration.

The assembly language System Configuration Module must be assembled as described in the toolset specific chapter of the AMX Tool Guide for inclusion in your AMX system.

If you are not using one of the toolsets supported by KADAK, you may still be able to use the AMX Configuration Builder by altering the System Configuration Template File as described in Appendix C.

The Configuration Generator is also available as a separate, stand alone DOS utility. This utility program can be used within your make files to generate and then compile your System Configuration Module. Instructions for doing so are provided in the AMX Tool Guide.

If you are not doing your development on a PC or compatible, you may still be able to port the Configuration Generator to your development system as described in Appendix C.

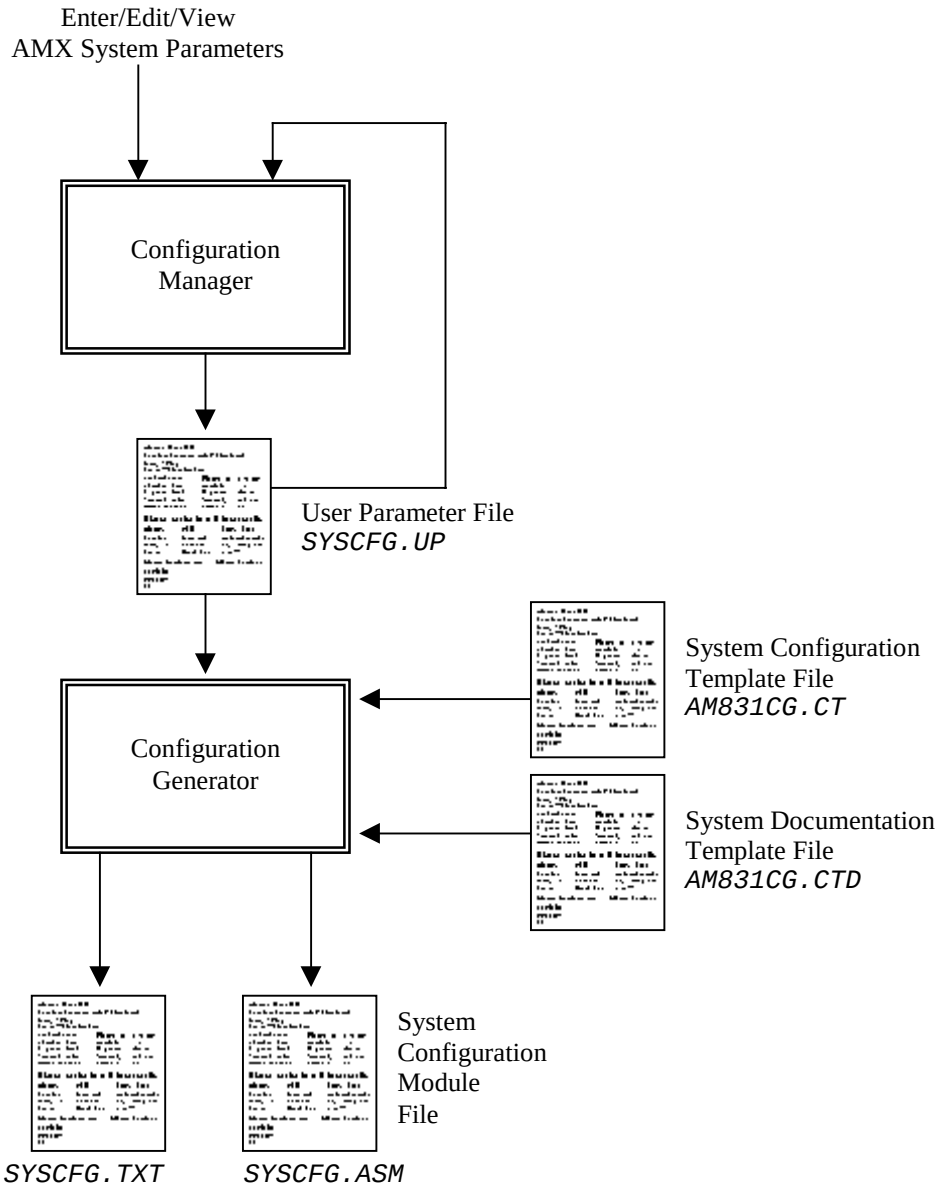


Figure 14.2-1 AMX Configuration Building Process

14.3 Using the Builder

Starting the Builder

The AMX Configuration Builder will operate on a PC or compatible running the Microsoft® Windows® operating system.

The Builder is delivered with the following files.

File	Purpose
AM831CM .EXE	AMX Configuration Manager (utility program)
AM831CM .CNT	AMX Configuration Manager Help Content File
AM831CM .HLP	AMX Configuration Manager Help File
AM831CG .EXE	AMX Configuration Generator (utility program)
AM831CG .CT	System Configuration Template File
AM831CG .CTD	System Documentation Template File

When AMX is installed on your hard disk, the AMX Configuration Manager for Windows utility program and its related files are stored in directory *CFGBLDW* in your AMX installation directory. To start the Configuration Manager, double click on its filename, *AM831CM.EXE*. Alternatively, you can create a Windows shortcut to the manager's filename and then simply double click the shortcut's icon.

To create a new User Parameter File, select New User Parameter File from the File menu. The Configuration Manager will create a new, as yet unnamed, file using its default AMX configuration parameters. When you have finished defining or editing your system configuration, select Save As... from the File menu. The Configuration Manager will save your User Parameter File in the location which you identify using the filename which you provide.

To open an existing User Parameter File, say *SYSCFG.UP*, select Open... from the File menu and enter the file's name and location or browse to find the file. When you have finished defining or editing your system configuration, select Save from the File menu. The Configuration Manager will rename your original User Parameter File to be *SYSCFG.BAK* and create an updated version of the file called *SYSCFG.UP*.

To direct the Configuration Manager to use its Configuration Generator utility to produce an updated copy of your System Configuration Module, say *SYSCFG.C*, select Generate... from the File menu.

The Configuration Manager can also be directed to use its Configuration Generator utility to produce a System Documentation Module, say *SYSCFG.TXT*, a text file summarizing the characteristics of your AMX configuration. Select Document... from the File menu.

Screen Layout

Figure 14.3-1 illustrates the Configuration Manager's screen layout. The title bar identifies the User Parameter File being created or edited. Below the title bar is the menu bar from which the operations you wish the Manager to perform can be selected. Below the menu bar is an optional Toolbar with buttons for many of the most frequently used menu commands.

At the bottom of the screen is the status bar. As you select menu items, a brief description of their purpose is displayed in the status bar. If the Configuration Manager encounters an error condition, it presents an error message on the status bar describing the problem and, in many cases, the recommended solution.

Along the left margin of the screen are a set of one or more selector icons. These icons identify the type of output files which the Manager's Configuration Generator will produce. The System Configuration Module selector must be active to generate the System Configuration Module and its related documentation file.

The center of the screen is used as an interactive viewing window through which you can view and modify your system configuration parameters.

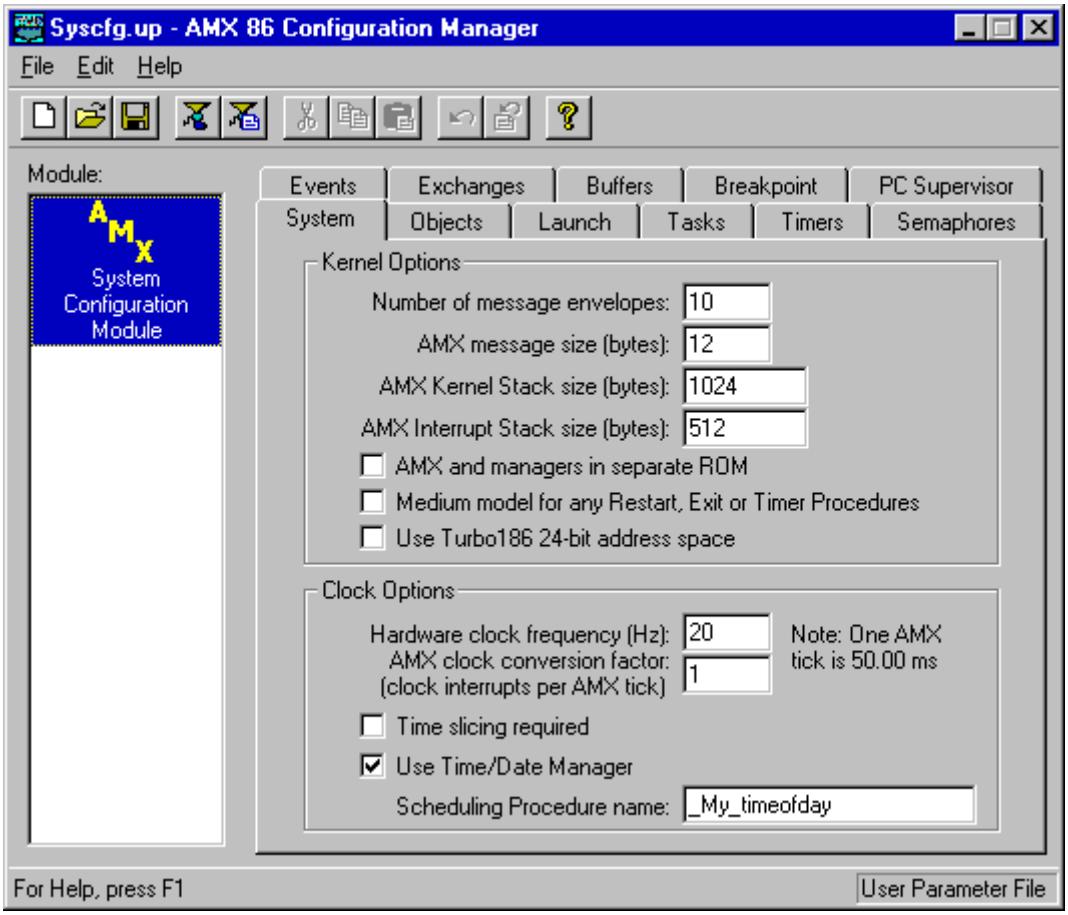


Figure 14.3-1 Configuration Manager Screen Layout

Menus

All commands to the Configuration Manager are available as items on the menus present on the menu bar. The **File menu** provides the conventional New, Open, Save and Save As... commands for creating and editing your User Parameter File. It also provides the Exit command.

When the System Configuration Module selector icon is the currently active selector, the Generate... command on the File menu can be used to generate your System Configuration Module and the File, Document... command can be used to generate your System Documentation Module. The paths to the template files required by the generator to create these products can be defined using the Templates... command on the File menu.

The **Edit menu** provides the conventional Cut, Copy, Paste and Undo editing commands. It also includes an Undo Page command to restore the content of all fields on a property page to undo a series of unwanted edits to the page. The Toolbar is hidden or made visible using the View Toolbar command on the Edit menu.

The **Help menu** provides access to the complete AMX Configuration Manager reference manual. Context sensitive help is also available by pressing the F1 function key or clicking the **?** button on the Toolbar.

Field Editing

When the System Configuration Module selector icon is the currently active selector, the System Configuration Module's tabbed property sheet is displayed in the central region of the screen. Each tab provides access to a particular property page through which your system configuration parameters can be declared. For instance, if you select the Tasks tab, the Configuration Manager will present a task definition window (property page) containing all of the parameters you must provide to completely define a task.

Some fields are boolean options in which all you can do is turn the option on or off by checking or unchecking the associated check box.

Some fields are option fields in which you must select one of a set of options identified with radio buttons. Click on the option button which meets your preference.

Other fields may require numeric or text entry. Parameters are entered or edited in these fields by typing new values or text to replace the current field content. Only displayable characters can be entered. New characters which you enter are inserted at the current cursor position in the field. Right and left arrow, backspace and delete keys may be used to edit the field.

When you are altering a numeric or text field, you tell the Configuration Manager that you are finished editing the field by striking the Enter key. At that point, the Configuration Manager checks the numeric value or text string that you have entered for correctness in the context of the current field. If the value or text string that you have entered is invalid, an error indication is provided on the status bar at the bottom of the screen suggesting how the fault should be corrected.

The Tab and Shift-Tab keys can also be used to complete the editing of a field and move to the next or previous field.

If you have modified some of the fields on a property page and then decide that these modified values are not correct, use the Undo Page command on the Edit menu or Toolbar to force the Configuration Manager to restore the content of all fields on the page to the values which were in effect when you moved to that property page.

When you go to save your User Parameter File or prepare to move to another property page, the Configuration Manager will validate all parameters on the page which you are leaving. If any parameters are incomplete or inconsistent with each other, you will be forced to fix the problem before being allowed to proceed.

Add, Edit and Delete AMX Objects

Separate property pages are provided to allow your definition of one or more AMX objects such as tasks and timers which will be prebuilt by AMX when AMX is launched. The maximum allowed quantities of each type of AMX object are defined by you on the Objects property page.

Pages of this type include a list box at the left side of the property page in which the currently defined objects are listed. At the bottom of the list box is a counter showing the number of objects in the list and the allowable maximum number of objects as defined on the Objects property page.

Also below the list are two control buttons labeled Add and Delete. If the allowable maximum number of objects is 0 or if all such objects have already been defined, the Add button will be disabled. If there are no objects defined, the Delete button and all other fields on the page will be disabled.

To add a new object, click on the Add button. A new object named - - - - will appear at the bottom of the list and will be opened ready for editing. When you enter a valid tag for the object, the tag will replace the name - - - - in the object list.

To edit an existing object's definition, double click on the object's name in the object list. The current values of all of that object's parameters will appear in the property page and the object will be opened ready for editing.

To delete an existing object, click on the object's name in the object list. Then click on the Delete button. Be careful because you cannot undo an object deletion.

The objects in the object list can be rearranged by dragging an object's name to the desired position in the list. You cannot drag an object directly to the end of the list. To do so, first drag the object to precede the last object on the list. Then drag the last object on the list to precede its predecessor on the list.

Tasks are ordered in their object list by task priority. The highest priority task resides at the top of the list. Tasks cannot be rearranged in their list by dragging the object. To do so, change the task's priority.

14.4 System Parameter Definition

The System Parameter window allows you to define the general operating parameters of your AMX system. The layout of the window is shown in Figure 14.3-1 in Chapter 14.3.

Kernel Options

AMX Message Envelopes

AMX passes a message to a mailbox or message exchange in a message envelope. The application parameters which form the message are copied into the envelope and added to the task mailbox or message exchange.

The number of message envelopes required by your application must be defined. The upper limit of the maximum number of envelopes is established by the fact that all message envelopes must reside in one segment which is limited to 64K bytes. For each of these message envelopes, the Builder will allocate storage for private AMX use.

Unfortunately, no hard and fast rule can be provided to determine the number of message envelopes needed. For simple systems with little message queuing and few tasks executing concurrently, the number of required message envelopes can range from one to twice the number of tasks in the system. More complex systems with significant queuing requirements may require many more envelopes.

AMX Message Size

The maximum size of the message to be passed in a message envelope must also be defined by you. The minimum message size is 12 bytes. The message size must be a multiple of 4.

AMX Kernel Stack Size

The AMX Kernel Task requires a minimum stack size of 128 bytes. In addition to this minimum, you must allocate sufficient stack to satisfy the worst case requirements of all application Restart Procedures and Timer Procedures. If you have any nonconforming ISPs, you must increase the Kernel Stack size to meet the interrupt overhead expected. No increase is needed if the ISP uses fewer than 40 bytes of stack. The stack size must be a multiple of 4 bytes.

AMX Interrupt Stack Size

The AMX Interrupt Supervisor requires a minimum Interrupt Stack size of 128 bytes. The stack size must be a multiple of 4 bytes.

In addition to this minimum, you must allocate sufficient stack to satisfy the application ISP with the greatest stack usage. If nested interrupts are possible, then the worst case interrupt nesting must be analyzed and additional interrupt stack size allocated. If *n* levels of nesting are permitted, then the interrupt stack size must be increased by *(64*n)+x* bytes, where *x* is the sum of the worst case stack use by each of the nested ISPs.

AMX and Managers in Separate ROM

AMX and its managers can be installed in a private ROM and accessed by your application via the AMX ROM access module. If you are using AMX this way, check this box.

If you are linking your application with AMX and its managers, leave this box unchecked even if you are going to burn your linked AMX system into a ROM.

Medium Model

If you have created any Medium model Restart Procedure, Exit Procedure or Timer Procedure, check this box. Otherwise leave this box unchecked.

24-bit Address Space

If you are using the Paradigm toolset (see AMX 86 Tool Guide, toolset *PX*) to create an AMX application for the VAutomation Turbo186 processor (or equivalent) with its 24-bit memory addressing, check this box. Otherwise leave this box unchecked.

Timing Options

Hardware Clock Frequency

This parameter defines the frequency of the AMX hardware clock in hertz. It is used by AMX to convert milliseconds to equivalent AMX system ticks. If your hardware clock frequency is not integral, round the clock frequency to the nearest non-zero integer.

AMX Clock Conversion Factor

The AMX system tick is measured in multiples of hardware clock interrupts. This parameter specifies the number of hardware clock interrupts required for each system tick. For example, a 100 Hz clock will generate interrupts at 10 millisecond intervals. If a 50 ms system tick is required, then this parameter must be set to 5.

Time Slicing Required

If you wish to allow tasks which share a priority level to be time sliced, check this box. Leave this box unchecked if time slicing of tasks is not required.

If time slicing has been enabled and a task has been defined to have a non-zero time slice, you will not be able to remove the check from this box. You must first set the time slice value for all defined tasks to 0. Then you will be able to remove the check from this box, thereby disabling time slicing.

Use Time/Date Manager

If you wish to include the Time/Date Manager, check this box. Otherwise leave the box unchecked. Be sure to increase the maximum number of timers in your system by one to account for the AMX Time/Date timer.

Scheduling Procedure Name

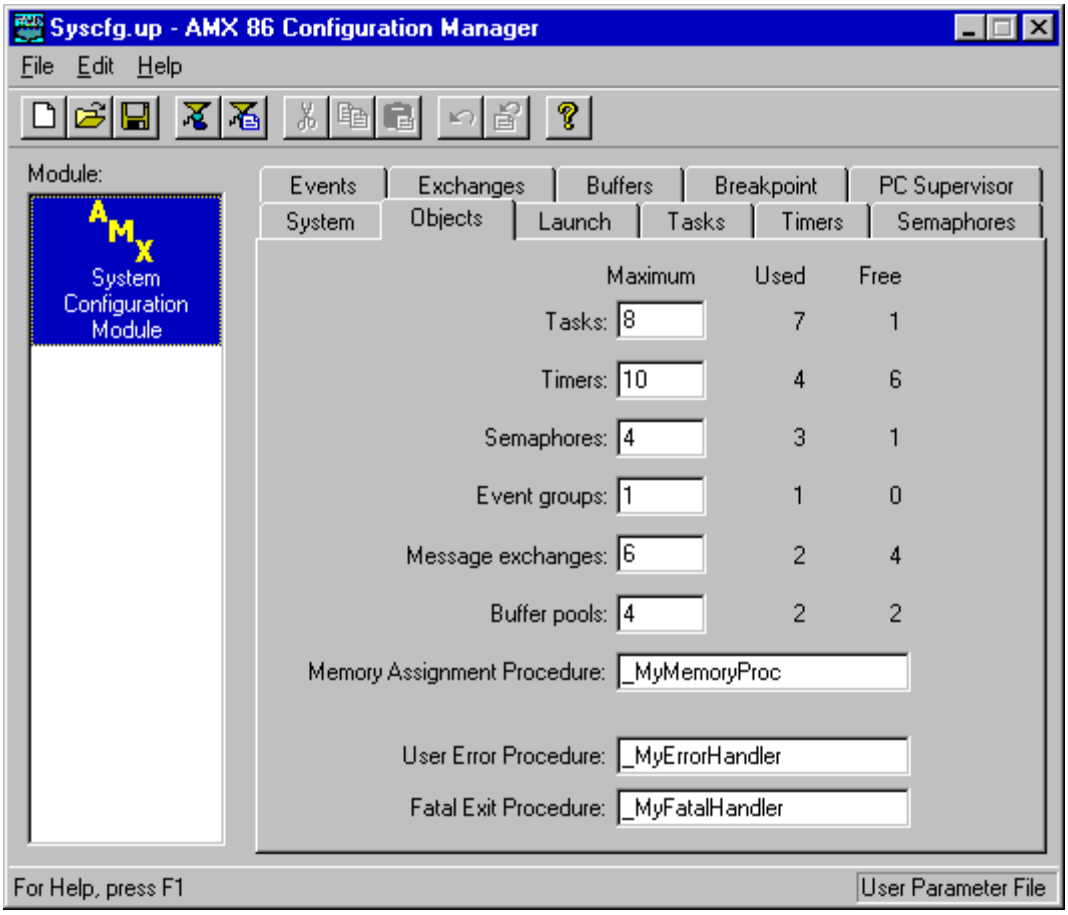
If you use the Time/Date Manager, then you may wish to define a Time/Date Scheduling Procedure (see Chapter 5.5). This parameter defines that procedure name. If no procedure is required, leave this field empty (blank).

This field is ignored if the Use Time/Date Manager check box is unchecked.

> Note
>
> When entering C procedure and variable names, be sure to add a leading underscore character (_) to the name if required for reference from assembly language to symbols generated by your C compiler.

14.5 AMX Object Allocation

The AMX Object Allocation window allows you to define the number of private AMX objects required for each of the optional AMX managers to be included in your system. The layout of the window is shown below.



Maximum Number of Tasks

This parameter defines the maximum number of application tasks which your system can support concurrently. The upper limit for this parameter is 100 tasks. The value for this parameter should be the number of predefined tasks (see Chapter 14.7) plus the maximum number of tasks that you will dynamically create using *ajtkcre*. You do not have to account for the AMX Kernel Task.

Maximum Number of Timers

This parameter defines the maximum number of timers which your system can support concurrently. If you do not require any timers, set this parameter to 0. Otherwise, this parameter should be set to the number of predefined timers (see Chapter 14.8) plus the maximum number of timers that you dynamically create using *ajtmcre*.

Maximum Number of Semaphores

This parameter defines the maximum number of resource and counting semaphores which your system can support concurrently. If you do not require any semaphores, set this parameter to 0. Otherwise, the parameter should be set to the number of predefined semaphores (see Chapter 14.9) plus the maximum number of resource or counting semaphores that you will dynamically create using *ajsmcre*.

Maximum Number of Event Groups

This parameter defines the maximum number of event groups which your system can support concurrently. If you do not require any event groups, set this parameter to 0. Otherwise, the parameter should be set to the number of predefined event groups (see Chapter 14.10) plus the maximum number of event groups that you will dynamically create using *ajevcre*.

Maximum Number of Message Exchanges

This parameter defines the maximum number of message exchanges which your system can support concurrently. If you do not require any message exchanges, set this parameter to 0. Otherwise, the parameter should be set to the number of predefined message exchanges (see Chapter 14.11) plus the maximum number of message exchanges that you will dynamically create using *ajmxcre*.

Maximum Number of Buffer Pools

This parameter defines the maximum number of buffer pools which your system can support concurrently. If you do not require any buffer pools, set this parameter to 0. Otherwise, the parameter should be set to the number of predefined buffer pools (see Chapter 14.12) plus the maximum number of buffer pools that you will dynamically create using *ajbcre*.

Memory Assignment Procedure Name

If you use the AMX Memory Manager, you must provide a Memory Assignment Procedure to assign memory sections to its memory pool for allocation to tasks (see Chapter 10.6). This parameter specifies the name of that procedure. If you are not using the AMX Memory Manager, leave this field empty (blank).

User Error Procedure Name

This parameter specifies the name of the User Error Procedure which AMX will call whenever any AMX service procedure posts an error indication (see Chapter 13.2). If you have no such procedure, leave this field empty (blank).

Fatal Exit Procedure Name

This parameter specifies the name of the Fatal Exit Procedure which is called by AMX whenever a fatal error condition is detected (see Chapter 13.1). If you have no such procedure, leave this field empty (blank).

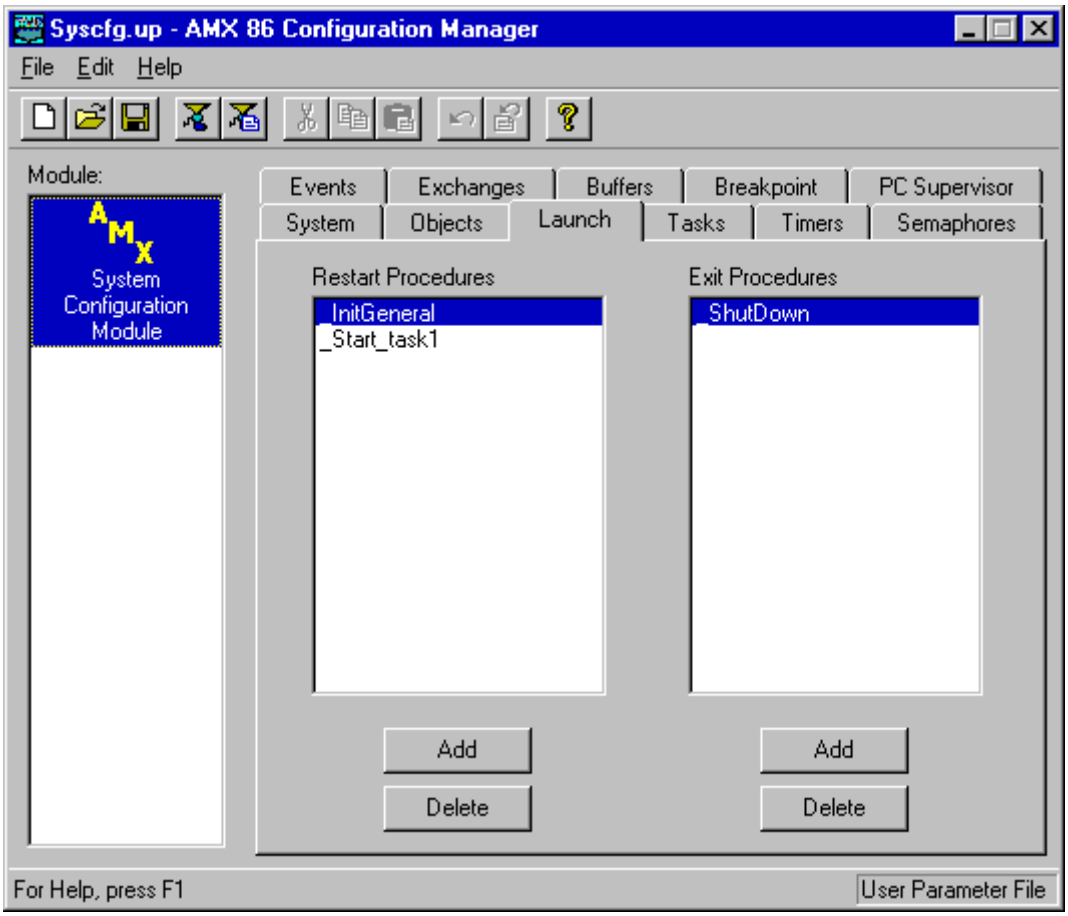
> Note
>
> When entering C procedure and variable names, be sure to add a leading underscore character (_) to the name if required for reference from assembly language to symbols generated by your C compiler.

14.6 Restart/Exit Procedure Definition

The Launch/Shutdown window displays all of your application Restart and Exit Procedures which will be called by AMX at system startup and shutdown. The layout of the window is shown below.

The Restart Procedure List is used to define all of your application Restart Procedures which will be called by AMX at system startup. The Restart Procedures will be called by AMX in the order in which they appear in the list.

The Exit Procedure List is used to define all of your application Exit Procedures which will be called by AMX at system shutdown. The Exit Procedures will be called by AMX in the order in which they appear in the list.



Add, Edit and Delete Restart and Exit Procedures

To add a new procedure, click on the Add button below the list. A new procedure named *---New---* will appear at the bottom of the list. Click on the name *---New---* and it will be opened ready for editing. Enter the name of your procedure.

To edit an existing procedure's name, double click on the name in the object list. The name will be opened ready for editing.

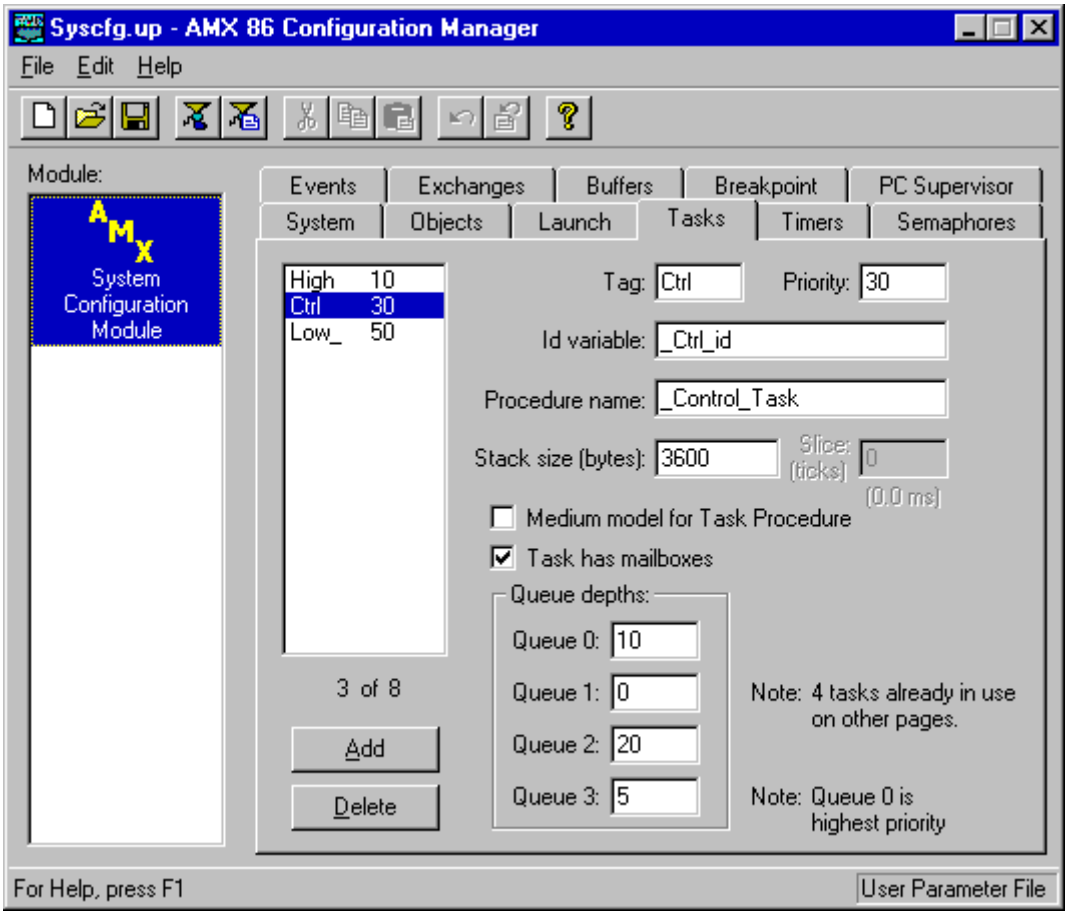
To delete an existing procedure, click on the procedure's name in the object list. Then click on the Delete button below the list. Be careful because you cannot undo a deletion.

The procedure names in the object list can be rearranged by dragging a procedure's name to the desired position in the list. You cannot drag a procedure directly to the end of the list. To do so, first drag the procedure name to precede the last name on the list. Then drag the last name on the list to precede its predecessor on the list.

> Note
>
> The Restart and Exit Procedures for all AMX Managers will be automatically provided by the Configuration Manager. You MUST NOT repeat them in your list of application Restart or Exit Procedures.

14.7 Task Definition

The Task Definition window allows you to define the tasks to be automatically created by AMX at system startup. You do not have to predefine all of your tasks in this manner; you may also create tasks dynamically using *ajtkcre*. The layout of the window is shown below.



Task Tag

Each task can have a unique 4-character task tag. This parameter defines that tag. Although AMX does not restrict the content of the task tag field, the Configuration Manager only supports 4 ASCII characters as a tag.

Priority

This parameter defines the execution priority of the task. Application task priorities range from 1 (highest) to 127 (lowest). More than one task can be given the same task priority. Unless the task is time sliced, AMX will assign relative task priorities according to the order in which the task definitions appear on the screen.

Id Variable

This parameter defines the name of a public variable of type *AMXID* in the System Configuration Module in which AMX will save the task id of the task.

Task Procedure Name

This parameter defines the name of the procedure to be called when AMX starts the task.

Stack Size

Each application task requires a block of RAM storage for use as a stack. The minimum stack size is 128 bytes. The stack size parameter must be a multiple of 4 bytes. If you increase the AMX message size beyond the minimum allowed (see Chapter 14.4), you must increase the size of all of your task stacks accordingly. In addition to this minimum, you must allocate sufficient stack to satisfy the worst case requirements of the task.

If your task installs its own task trap handlers (see Chapter 4.5), increase your task's stack by 64 bytes.

If you have any nonconforming ISPs, you must increase the stack size of all tasks to meet the needs of these ISPs. No increase is needed if the ISP uses fewer than 40 bytes of stack.

Time Slice

If the task is to be time sliced, you may specify the time slice interval in the task definition. If time slicing is not required, set this parameter to 0.

This field will be disabled unless you have enabled the AMX time slice option by checking the Time Slicing Required box on the System Parameter Window.

Time slicing is only effective if the task will be compute bound in competition with other tasks of equal priority.

The task's time slice may also be altered dynamically by calling *ajtslv* when your system is running.

Medium Model

Tasks can be either Large or Medium. If your task is Medium model, check this box. Otherwise, leave this box unchecked.

If you declare the task to be Medium, the Builder will allocate the task's stack in a segment which is part of group *DGROUP* so that *DS = SS = DGROUP* when the task executes.

Task Message Queues (Mailboxes)

AMX allows creation of a task with a set of four private message queues (mailboxes) on which the task can receive AMX messages. To declare a task of this type, check this box and fill in the required queue depths for the task's private message queues. Otherwise leave the box unchecked.

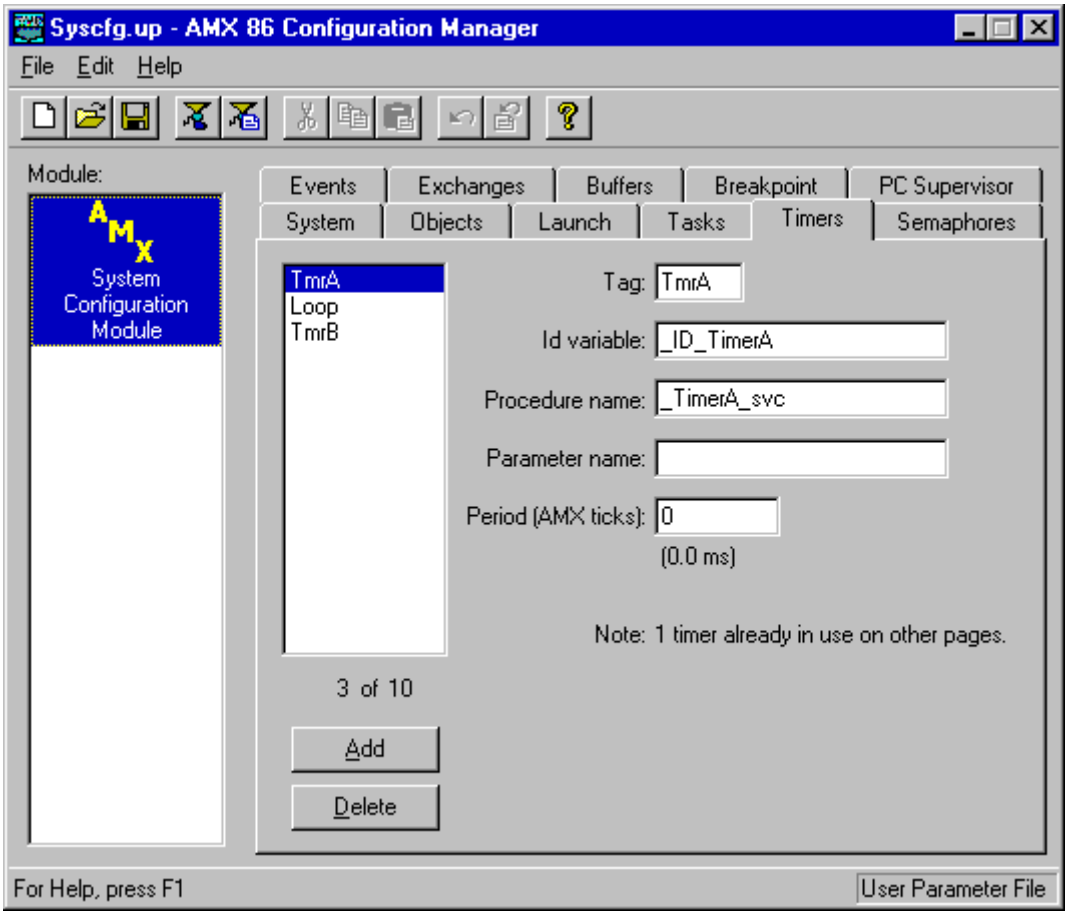
Queue Depths

These four parameters define the maximum number of message envelopes which can reside in each of the task's four private message queues. Queue 0 is the highest priority queue; queue 3 is the lowest priority.

Depths may range from 0 to 32767. The maximum depth of a queue does not affect AMX memory requirements. If a particular message queue is to be unused, set its depth to zero. At least one message queue depth must be non-zero if the task is to be defined as requiring message queues.

14.8 Timer Definition

The Timer Definition window allows you to define the timers to be automatically created by AMX at system startup. You do not have to predefine all of your timers in this manner; you may also create timers dynamically using *ajtmcre*. Note that AMX does not automatically start timers defined this way. Your application must call *ajtmwr* to actually start the timer. The layout of the window is shown below.



Tag

Each timer can have a unique 4-character timer tag. This parameter defines that tag. Although AMX does not restrict the content of the timer tag field, the Configuration Manager only supports 4 ASCII characters as a tag.

Id Variable

This parameter defines the name of a public variable of type *AMXID* in the System Configuration Module in which AMX will save the timer id of the timer.

Timer Procedure Name

This parameter defines the name of the Timer Procedure to be called whenever the timer expires.

Timer Parameter Name

This parameter defines the name of a public pointer variable whose address will be passed as a parameter to the Timer Procedure. If your timer does not require such a parameter, leave this field empty (blank).

Assume that your application includes a C variable *tmparam* declared as follows:

```
void *tmrparam;
```

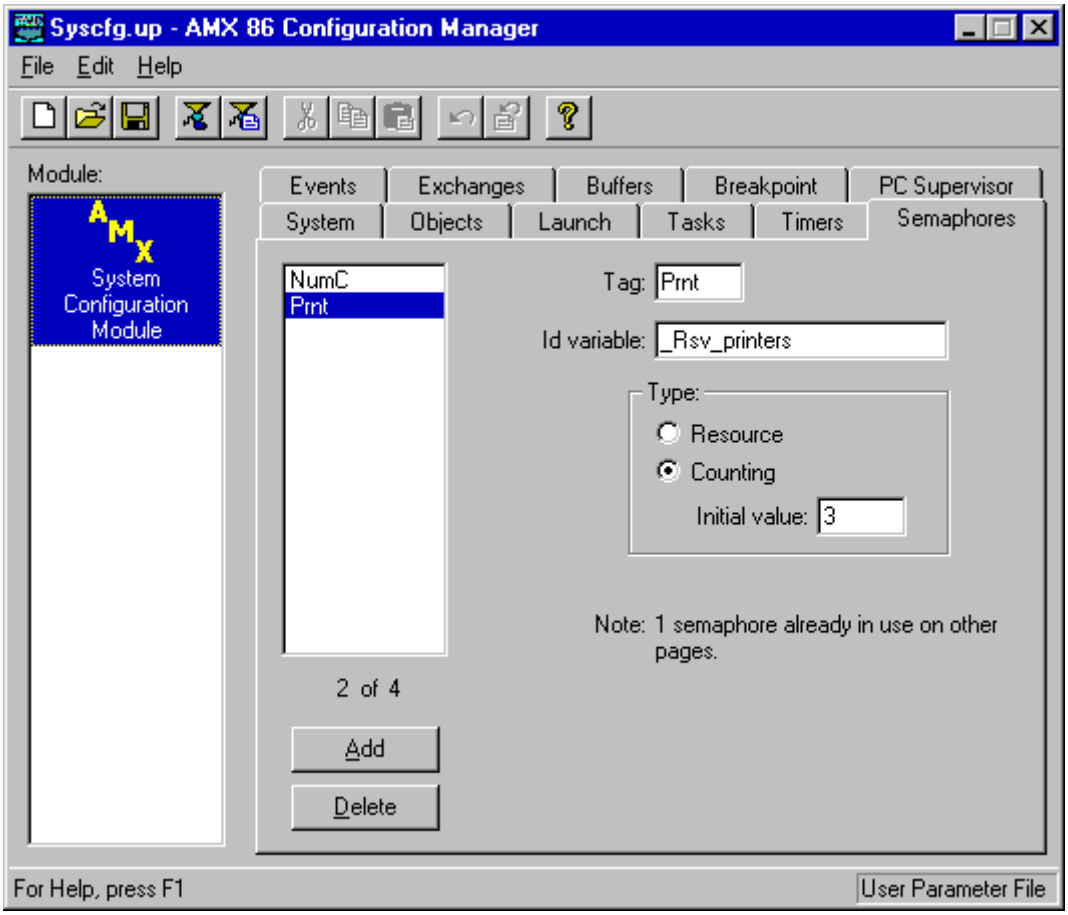
Enter the name *_tmrparam* as the name of the timer parameter. Your Timer Procedure, coded in C, will then receive *&tmrparam* as its parameter.

Period

If the timer is a periodic timer, this parameter defines its period in multiples of AMX system ticks. If the timer is a one-shot timer, this parameter must be set to 0.

14.9 Semaphore Definition

The Semaphore Definition window allows you to define the semaphores to be automatically created by AMX at system startup. You do not have to predefine all of your semaphores in this manner; you may also dynamically create resource semaphores and counting semaphores using *ajsmcre*. The layout of the window is shown below.



Tag

Each semaphore can have a unique 4-character semaphore tag. This parameter defines that tag. Although AMX does not restrict the content of the semaphore tag field, the Configuration Manager only supports 4 ASCII characters as a tag.

Id Variable

This parameter defines the name of a public variable of type *AMXID* in the System Configuration Module in which AMX will save the semaphore id of the semaphore.

Type

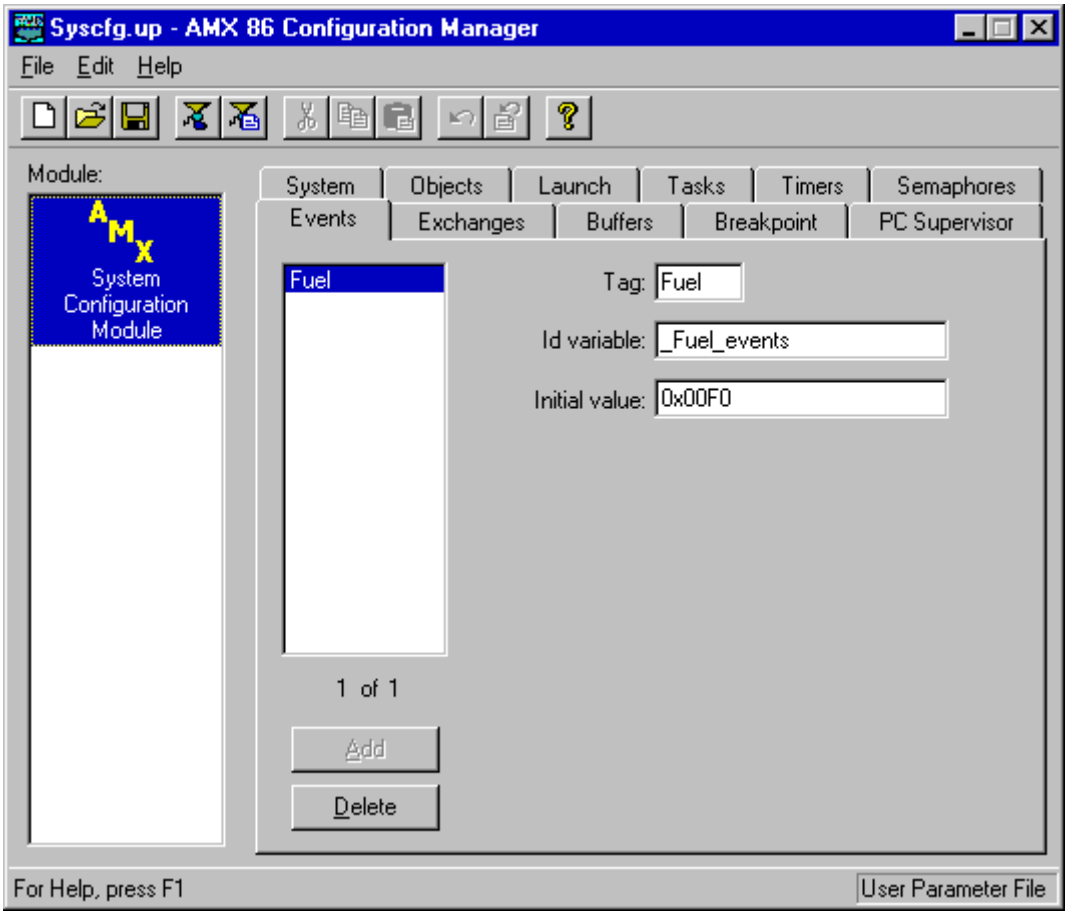
This option field defines the type of semaphore to be created, resource or counting. The initial value field is ignored for resource semaphores.

Initial Value

Counting semaphores may be given an initial value between 0 and 32767.

14.10 Event Group Definition

The Event Group Definition window allows you to define the event groups to be automatically created by AMX at system startup. You do not have to predefine all of your event groups in this manner; you may also create event groups dynamically using *ajevcre*. The layout of the window is shown below.



Tag

Each event group can have a unique 4-character event group tag. This parameter defines that tag. Although AMX does not restrict the content of the event group tag field, the Configuration Manager only supports 4 ASCII characters as a tag.

Id Variable

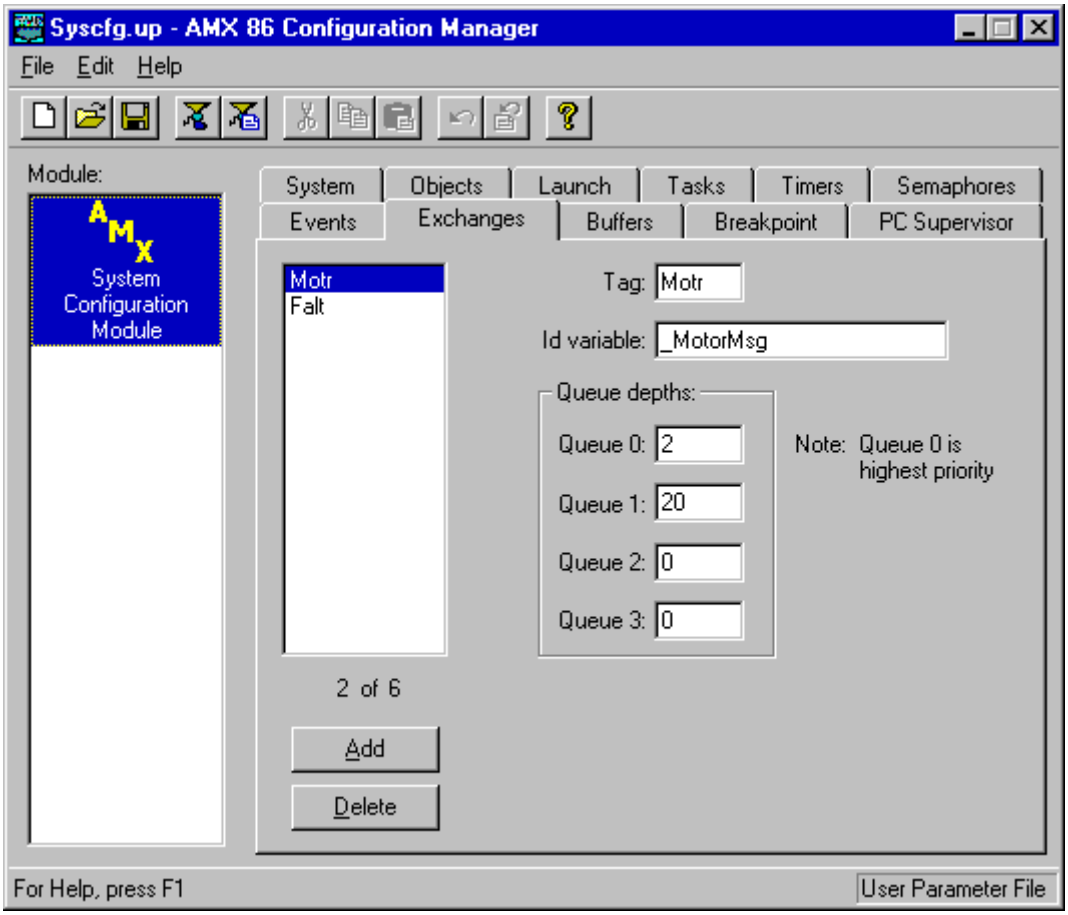
This parameter defines the name of a public variable of type *AMXID* in the System Configuration Module in which AMX will save the event group id of the event group.

Initial Value

This parameter defines the initial value of the 16 event flags in the event group. This parameter must be entered as 0 or as an unsigned hexadecimal number of the form *0xdddd*.

14.11 Message Exchange Definition

The Message Exchange Definition window allows you to define the message exchanges to be automatically created by AMX at system startup. You do not have to predefine all of your message exchanges in this manner; you may also create message exchanges dynamically using *ajmxcre*. The layout of the window is shown below.



Tag

Each message exchange can have a unique 4-character message exchange tag. This parameter defines that tag. Although AMX does not restrict the content of the message exchange tag field, the Configuration Manager only supports 4 ASCII characters as a tag.

Id Variable

This parameter defines the name of a public variable of type *AMXID* in the System Configuration Module in which AMX will save the message exchange id of the message exchange.

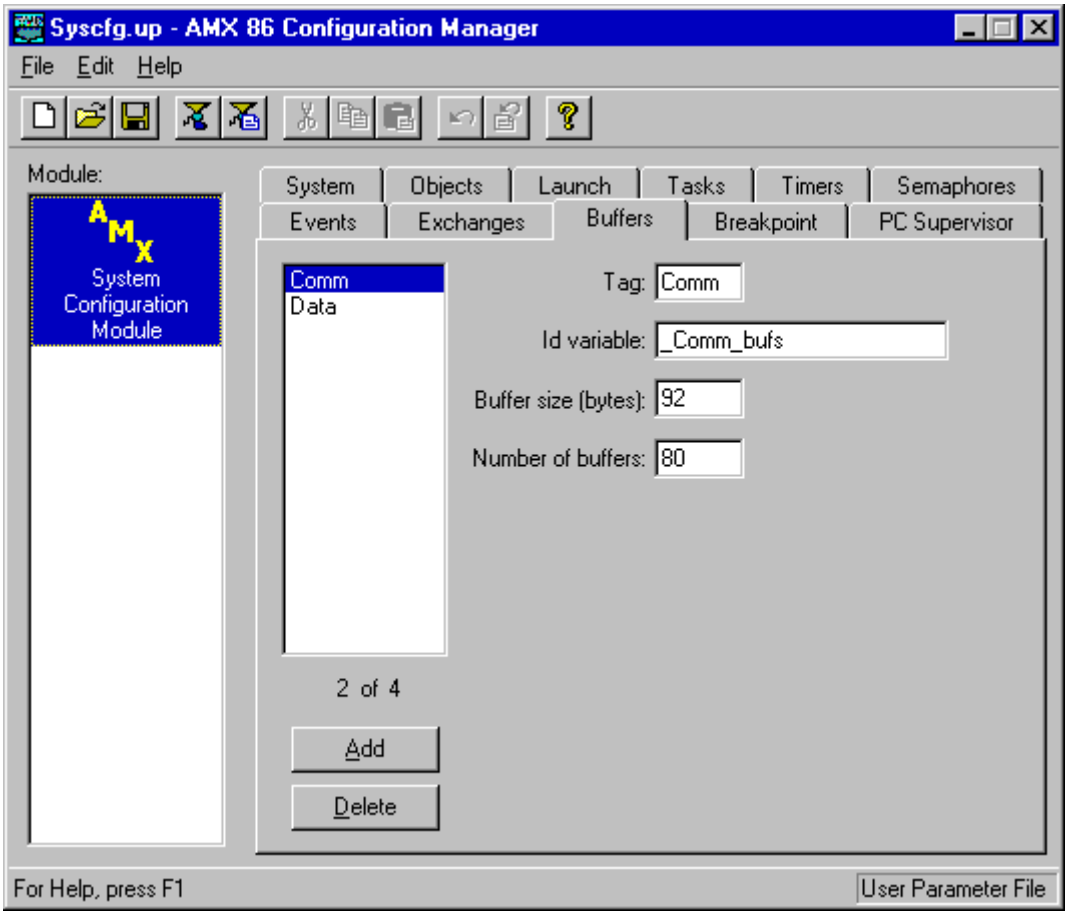
Message Queue Depths

These four parameters define the maximum number of message envelopes which can reside in each of the four message queues of the message exchange. Message queue 0 is the highest priority queue; queue 3 is the lowest priority.

Depths may range from 0 to 32767. If a particular message queue for a message exchange is to be unused, set its depth to zero. Other message queues may still be used. At least one message queue must be used. The maximum depth for a message queue does not affect AMX memory requirements.

14.12 Buffer Pool Definition

The Buffer Pool Definition window allows you to define the buffer pools to be automatically created by AMX at system startup. You do not have to predefine all of your buffer pools in this manner; you may also create buffer pools dynamically using *ajbcre*. The layout of the window is shown below.



Tag

Each buffer pool can have a unique 4-character buffer pool tag. This parameter defines that tag. Although AMX does not restrict the content of the buffer pool tag field, the Configuration Manager only supports 4 ASCII characters as a tag.

Id Variable

This parameter defines the name of a public variable of type *AMXID* in the System Configuration Module in which AMX will save the buffer pool id of the buffer pool.

Buffer Size

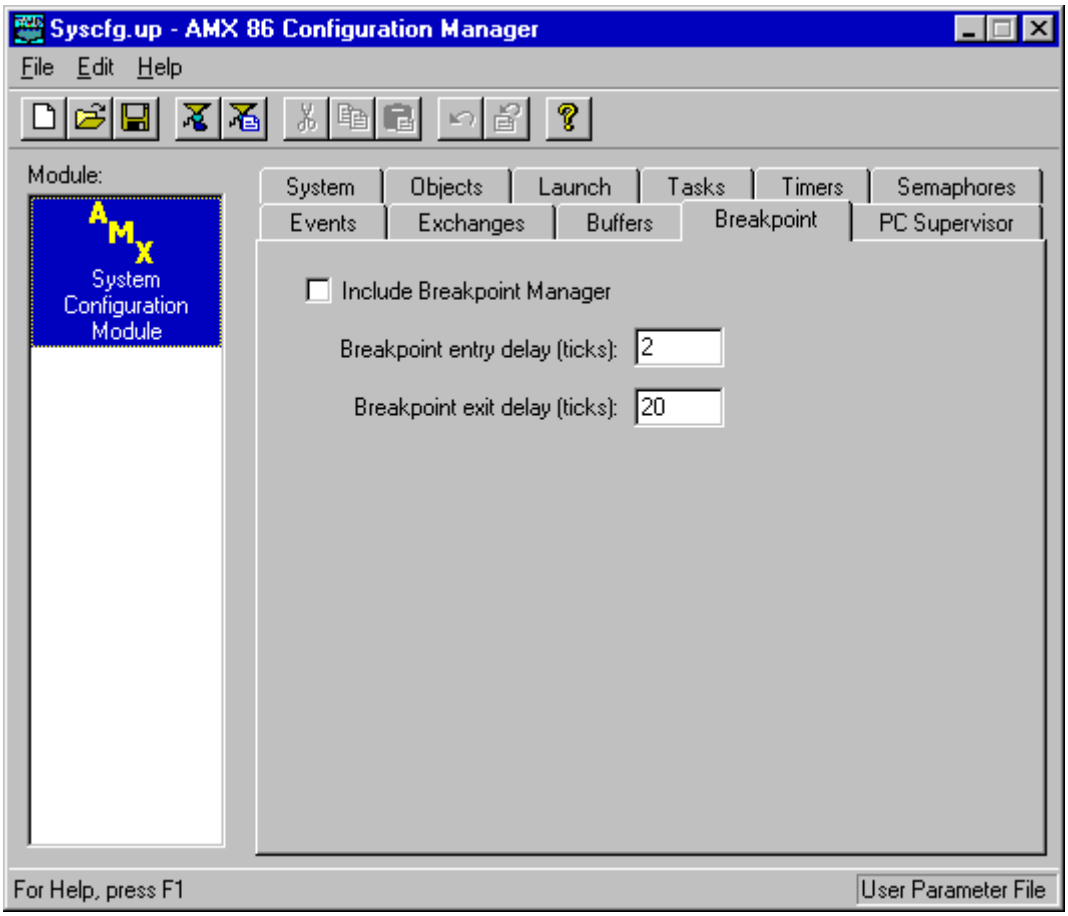
This parameter defines the useable size (in bytes) of each buffer in the buffer pool. The minimum buffer size is 8 bytes. The buffer size must be a multiple of 4.

Number of Buffers

This parameter defines the total number of buffers to be available in this buffer pool. The pool must contain at least one buffer.

14.13 Breakpoint Manager Definition

The AMX Breakpoint Manager can be used to improve the operation of your debugger in the AMX multitasking environment. Your use of the Breakpoint Manager is optional. The layout of the window is shown below.



Include Breakpoint Manager

If you need to use the AMX Breakpoint Manager to shut off all task activity when your debugger hits a breakpoint, check this box. Otherwise, leave the box unchecked.

Entry/Exit Delay

If your debugger inhibits clock interrupts at breakpoints, you can improve the Breakpoint Manager's exit detection as follows. Set the breakpoint entry delay to *-1* and the breakpoint exit delay to *1*. The Breakpoint Manager will assume that you have left the breakpoint as soon as the next AMX clock tick is detected.

Otherwise, set both of these parameters to *0* or to the values specified in the toolset dependent chapters of the AMX 86 Tool Guide.

This page left blank intentionally.

15. AMX Service Procedures

15.1 Introduction

The AMX Library provides a wide variety of services from which the real-time system designer can choose. Many of the services are optional and, if not used, will not even be present in your final AMX system.

This section of the AMX User's Guide differs from the others because it is intended for use as a programming guide. The remainder of this chapter introduces you to the AMX programming environment. Chapter 16 provides descriptions of all of the procedures which are available in the AMX Runtime Library in alphabetic order for easy reference.

All of the AMX procedures are described using the C programming language. It is therefore recommended that you be at least superficially familiar with C.

A functional list of procedures is presented in Chapter 15.2. It is recommended that you use that chapter in conjunction with the procedure descriptions in Chapter 16 as follows. If you remember a procedure name but require information concerning its operation, go straight to the procedure description in the alphabetic list of procedures in Chapter 16. If you know what you wish to do functionally, but cannot remember the procedure name, go to Chapter 15.2 to quickly locate the procedure and then proceed to Chapter 16 to find its detailed description.

15.2 Summary of Services

AMX provides a wide variety of services from which the real-time system designer can choose. Many of the services are optional and, if not used, will not even be present in your AMX system. The AMX managers are all optional.

All of AMX and its managers are fully reentrant and may be placed in Read Only Memory (ROM).

The following lists summarize all of the AMX procedures which are accessible to the user. They are grouped functionally for easy reference.

Procedures are described in Chapter 16.

System Control Kernel Services

ajentr	*AAENTR*	Enter AMX multitasking world
ajexit	*AAEXIT*	Leave AMX multitasking world
ajfatl	*AAFATL*	Fatal exit
ajhook	*AAHOOK*	Install user hooks to AMX Scheduler
ajprvl	*AAPRVL*	Lower privilege level
ajprvr	*AAPRVR*	Raise privilege level
ajupt		Find User Parameter Table
ajver	*AAVER*	Get AMX version number

Interrupt Control for AMX 86

ajint	*AAINT*	Begin interrupt service
ajinx	*AAINX*	End interrupt service
ajispm	*AAISPM*	Make an ISP root
ajitrp	*AAITRP*	Install task trap handler
ajivtr	*AAIVTR*	Read entry in IVT
ajivtw	*AAIVTW*	Write entry in IVT
ajivtx	*AAIVTX*	Exchange entry in IVT

Task Control

ajend	*AAEND*	End task execution
ajgmsg	*AAGMSG*	Get the highest priority message available Optionally get a message of a specific priority
ajresum	*AARESUM*	Resume a suspended task
ajsend *ajsendp*	*AASEND*	Start a task by sending it a message at one of 4 priorities
ajsenw *ajsenwp*		Optionally wait for that task to receive the message
ajsgnl	*AASGNL*	Signal a task
ajsgrd	*AASGRD*	Read pending task signals
ajsgres	*AASGRES*	Reset pending task signals
ajsgwat	*AASGWAT*	Wait for task signals
ajsusp	*AASUSP*	Suspend a task
ajtkcre	*AATKCRE*	Create a new task
ajtkid	*AATKID*	Get task identifier of current task
ajtkpry	*AATKPRY*	Change a task's execution priority
ajtksts	*AJTKSTS*	Fetch task status
ajtktag	*AATKTAG*	Find task id of task with a specific task tag
ajtktcb	*AATKTCB*	Fetch pointer to a Task Control Block
ajtrig	*AATRIG*	Start (trigger) a task with no message
ajwait	*AAWAIT*	Wait for a wake request
ajwakc	*AAWAKC*	Wake a task upon receipt of its message
ajwakcs	*AAWAKCS*	Wake a task upon receipt of its message; return status to that task
ajwake	*AAWAKE*	Wake a task which is waiting
ajwapr	*AAWAPR*	Reset pending wake for current task
ajwatm	*AAWATM*	Wait for a timed interval or until a wake request occurs

Task Termination Services

ajtkdel	*AATKDEL*	Delete a task
ajtkill	*AATKILL*	Kill a task by forcing it to stop; flush its message mailboxes
ajtkstp	*AATKSTP*	Force a task to stop
ajtktrm	*AATKTRM*	Enable/disable abnormal task termination

Timing Control

ajclk	*AACLK*	AMX Clock Handler
ajtick	*AATICK*	Read elapsed system ticks
ajtmcnv	*AATMCNV*	Convert milliseconds to system ticks
ajtmcre	*AATMCRE*	Create an interval timer
ajtmdel	*AATMDEL*	Delete an interval timer
ajtmrd	*AATMRD*	Read an interval timer
ajtmtag	*AATMTAG*	Find timer id of timer with a specific tag
ajtmwr	*AATMWR*	Start/stop an interval timer
ajtslv	*AATSLV*	Change a task's time slice interval
ajtsof	*AATSOF*	Disable time slicing
ajtson	*AATSON*	Enable time slicing

Time/Date Manager

ajtdf	*AATDF*	Format calendar time/date as an ASCII string
ajtdg	*AATDG*	Get calendar time/date
ajtds	*AATDS*	Set calendar time/date

Semaphore Manager

ajsmcre	*AASMCRE*	Create a semaphore
ajsmdel	*AASMDEL*	Delete a semaphore
ajsmfre	*AASMFRE*	Free a resource (unconditional)
ajsmget	*AASMGET*	Get use of a semaphore (no wait)
ajsmrls	*AASMRLS*	Release a resource (nested)
ajsmrsv	*AASMRSV*	Reserve a resource (optional timeout)
ajsmsig	*AASMSIG*	Signal to a semaphore
ajsmtag	*AASMTAG*	Find semaphore id of semaphore with a specific tag
ajsmwat	*AASMWAT*	Wait on a semaphore (optional timeout)

Event Manager

ajevcre	*AAEVCRE*	Create an event group
ajevdel	*AAEVDEL*	Delete an event group
ajevnt	*AAEVNT*	Get saved event flags
ajevrd	*AAEVRD*	Read current state of events in a group
ajevsig	*AAEVSIG*	Signal one or more events in a group
ajevtag	*AAEVTAG*	Find group id of event group with a specific tag
ajevwat	*AAEVWAT*	Wait for all/any of a set of events in a group (optional timeout)

Message Exchange Manager

ajmxcre	*AAMXCRE*	Create a message exchange
ajmxdel	*AAMXDEL*	Delete a message exchange
ajmxget	*AAMXGET*	Get a message from a message exchange (no wait)
ajmxsnd *ajmxsndp*	*AAMXSND*	Send message to a message exchange
ajmxtag	*AAMXTAG*	Find exchange id of message exchange with a specific tag
ajmxwat	*AAMXWAT*	Wait for message to arrive at a message exchange (optional timeout)

Buffer Manager

ajbau	*AABAU*	Add to buffer use count
ajbcre	*AABCRE*	Create a buffer pool
ajbdel	*AABDEL*	Delete a buffer pool
ajbfre	*AABFRE*	Free a buffer
ajbget	*AABGET*	Get a buffer from a specific pool
ajbgsz	*AABGSZ*	Get size of buffer
ajbia	*AABIA*	Initialize all currently defined buffer pools
ajbip	*AABIP*	Initialize one buffer pool
ajbtag	*AABTAG*	Find pool id of a buffer pool with a specific tag

Memory Manager

ajmau	*AAMAU*	Add to block use count
ajmfre	*AAMFRE*	Free a block of memory
ajmgeh	*AAMGEH*	Get a block of memory from a private memory section
ajmget	*AAMGET*	Get a block of memory
ajmgsz	*AAMGSZ*	Get size of block of memory
ajmhan	*AAMHAN*	Create a handle to a private memory section
ajmset	*AAMSET*	Fill a block of memory with a pattern

Circular List Manager

ajabl	*AAABL*	Add to bottom of circular list
ajatl	*AAATL*	Add to top of circular list
ajrbl	*AARBL*	Remove from bottom of circular list
ajrstl	*AARSTL*	Reset a circular list
ajrtl	*AARTL*	Remove from top of circular list

Linked List Manager

ajlcre	*AALCRE*	Create an empty list
ajlhead	*AALHEAD*	Find head of a list
ajlinsc	*AALINSC*	Insert before the specified object on a list
ajlinsh	*AALINSH*	Insert at the head of a list
ajlinsk	*AALINSK*	Insert into a keyed list
ajlinst	*AALINST*	Insert at the tail of a list
ajlmerg	*AALMERG*	Merge two lists
ajlnext	*AALNEXT*	Find next object on a list (walk towards tail)
ajlordk	*AALORDK*	Reorder object on a keyed list
ajlprev	*AALPREV*	Find previous object on a list (walk towards head)
ajlrmvc	*AALRMVC*	Remove specified object from a list
ajlrmvh	*AALRMVH*	Remove object from the head of a list
ajlrmvt	*AALRMVT*	Remove object from the tail of a list
ajltail	*AALTAIL*	Find tail of a list

Processor and C Interface Procedures

In addition to the services provided by AMX and its managers, the AMX Library includes several C procedures of a general nature which simplify application programming in real-time systems on your target processor.

ajcfjlong	Long jump to a mark set by *ajcfjset*
ajcfjset	Set a mark for a subsequent long jump by *ajcfjlong*
ajcfstkjmp	Switch stacks and jump to a new procedure
ajdi	Disable interrupts
ajei	Enable interrupts
ajflagrd	Read the processor flags register (*FLAGS*)
ajflagrddi	Read the processor flags register (*FLAGS*); then disable interrupts
ajflagwr	Write to the processor flags register (*EFLAGS*)
ajgofs	Get *FAR* pointer offset
ajgseg	Get *FAR* pointer selector
ajgsreg	Get (read) all segment registers
ajmodl	Set *DS* to the *DGROUP* segment value
ajproc	Generate assembler procedure *CALL*
ajprocq	
ajsint	Generate software interrupt
ajsintq	
ajsofs	Set *FAR* pointer offset
ajsseg	Set *FAR* pointer selector
ajssreg	Set all segment registers (except *CS*, *SS*)
	Hardware I/O port operations
ajinb	Read from an 8-bit input I/O port
ajinw	Read from a 16-bit input I/O port
ajoutb	Write an 8-bit value to an output I/O port
ajoutw	Write a 16-bit value to an output I/O port

AMX 86 PC Supervisor Service Procedures

In addition to the services provided by AMX and its managers, the AMX 86 PC Supervisor Library includes several C procedures which are of use only if the AMX 86 PC Supervisor is used by your application. These procedures are documented in Chapter 3.9 of the AMX 86 PC Supervisor Reference Manual.

ajbeep	*AABEEP*	Beep speaker
ajboot	*AABOOT*	Boot DOS
ajkbgc	*AAKBGC*	Get keyboard character
ajkbof	*AAKBOF*	Clear specific keyboard features
ajkbon	*AAKBON*	Set specific keyboard features
ajkbss	*AAKBSS*	Sense keyboard status
ajpdrl	*AAPDRL*	Release DOS
ajpdrp	*AAPDRP*	Reserve DOS (with priority)
ajpdrs	*AAPDRS*	Reserve DOS
ajspof	*AASPOF*	Turn speaker off
ajspon	*AASPON*	Turn speaker on

16. AMX 86 Procedures

16.1 Introduction

A description of every AMX Library procedure is provided in this chapter. The descriptions are ordered alphabetically for easy reference.

Italics are used to distinguish programming examples. Procedure names and variable names which appear in narrative text are also displayed in italics. Occasionally a lower case procedure name or variable name may appear capitalized if it occurs as the first word in a sentence.

Vertical ellipses are used in program examples to indicate that a portion of the program code is missing. Most frequently this will occur in examples where fragments of application dependent code are missing.

```
:
: /* Dismiss device interrupt */
:
```

Capitals are used for all defined AMX filenames, constants and error codes. All AMX procedure, structure and constant names can be readily identified according to the nomenclature introduced in Chapter 1.3.

A consistent style has been adopted for each description. The procedure name is presented at the extreme top right and left as in a dictionary. This method of presentation has been chosen to make it easy to find procedures since they are ordered alphabetically.

Purpose A one-line statement of purpose is always provided.

Used by ■ Task □ ISP □ Timer Procedure ■ Restart Procedure □ Exit Procedure

This block is used to indicate which of your AMX application procedures can call the AMX procedure. The term ISP refers to the Interrupt Handler of a conforming ISP. A filled in box indicates that the procedure is allowed to call the AMX procedure. In the above example, only tasks and Restart Procedures would be allowed to call the procedure.

Setup Defines all input parameters to the procedure and gives an example of the calling sequence.

If you are programming in C, you must include AMX header files *AMX831SD.H* and *AMX831EC.H* so that you will have access to all structure, constant and error code definitions. File *AMX831SD.H* automatically includes AMX header file *AMX831CF.H* to give you access to all AMX function prototypes. Within this chapter, header files are not shown in the setup unless the function references an AMX structure or constant.

If you are programming in assembly language, you must include AMX header files *AMX831SD.DEF* and *AMX831EC.DEF* so that you will have access to all structure, constant and error code definitions. In your modules, AMX functions must be declared as external *FAR* procedures.

Where If the procedure has input parameters, they will be described narratively.

Results The outputs produced by the procedure are always defined.

The state of the carry, parity, auxiliary, zero, sign and overflow flags are of no consequence when programming in C. In assembly language, the zero and sign flags reflect the value of the error code *AERxxxx*, if any, returned by AMX in register *AX*. The direction flag, if used, is always restored to its original state.

AMX procedures frequently must deal with the processor interrupt enable flag. The state of this flag upon exit from each AMX procedure is defined.

All other 8086 processor flags are untouched by AMX.

Used By Some procedures such as *ajentr* cannot be called by any of your application procedures listed at the top right corner of the page. The allowable caller is explicitly identified.

Restrictions If any restrictions on the use of the procedure exist, they are described.

Note Special notes, suggestions or warnings are offered where necessary.

See Also A cross reference to other related AMX procedures is always provided if applicable.

All AMX procedures are coded as *FAR* procedures. Be sure that your C procedures follow Medium or Large model conventions when calling AMX procedures.

Almost all AMX procedures only return integers, the default assumed by the C language.

All AMX procedures assume that an integer or unsigned integer is a 16-bit word value. This assumption is consistent with the conventions adopted by most currently available 16-bit C compilers for the 80x86.

Assembly Language Programming

If you are programming in assembly language, refer to Appendix E for a description of the AMX assembly language calling conventions. The appendix includes a summary of the input and output register specifications for every AMX procedure. Experienced AMX programmers will usually find that the appendix is all that is required as a programmer's reference guide.

New AMX users will want to reference the detailed AMX procedure descriptions provided in this chapter. For their benefit, the assembly language input and output register specifications have been included in a cryptic form along with the C language call specifications.

Notational Conventions

The following example illustrates the notational conventions used for assembly language calling sequences.

```
AMXID    semid;
int      value;
char     tag[4];
int      status;
 .
 .
status = ajsmcre(&semid, value, tag);
AX               BX=     BX     [DX:CX]
```

Let *reg* be the assembly language mnemonic for any general 8 or 16-bit 8086 register. Let *seg* be the assembly language mnemonic for any 8086 segment register.

If *reg* appears underneath any input or output parameter name, then the value of that parameter is presented or received in register *reg*. In the example, *value* is presented to assembly language procedure *AASMCRE* in *BX* and *status* is received from it in *AX*.

Some AMX procedures must return more than a single integer result to the caller. When programming in C, the caller must provide a pointer to a variable into which AMX can store the result (*&semid* for example). The syntax *reg=* is used to indicate that at the assembler level, the result is returned in register *reg*, not in storage referenced by *reg*. In the example, the value which would be assigned to C variable *semid* is received from *AASMCRE* in register *BX*.

Occasionally, parameters are passed by reference in C but by value in assembly language. For example, 4-character tags are passed as string pointers in C but are passed in a register pair in assembly language. The syntax *[reg]* underneath a pointer to a parameter indicates that the actual parameter referenced by the pointer is passed in register *reg*. In the example, the four bytes referenced by pointer *tag* are passed to *AASMCRE* in register *DX:CX*.

FAR pointers are passed to AMX as *seg:reg* as in the following example.

```
status = ajsenw(taskid, priority, (void FAR *)msg);
AX       AASEND DX      CX[14..0]             ES:BX
                        CX[15]=1
```

Specific bits in a register *reg* are referenced using the syntax *reg[bit]* or *reg[highbit..lowbit]*. In this example, *CX[15]* is bit 15 of register *CX*. *CX[14..0]* is bits 0 to 14 inclusive of register *CX*. Unspecified bits in a register must be 0.

The above example also illustrates that procedure *AASEND* is the assembly language counterpart of *ajsenw*, not *AASENW* as might otherwise be expected.

One final notational syntax is illustrated by the following example.

```
status = ajmxsndp(exchange, priority, parm1, parm2...);
AX       AAMXSND  BX        DX        @ES:SI
```

Procedure *AAMXSND* requires a *FAR* pointer to an AMX message in *ES:SI*. The message is on the stack as *parm1, parm2...*. The syntax *@seg:reg* underneath a parameter means that *seg:reg* will contain a *FAR* pointer to that parameter on the stack. In this example, *ES:SI* will point to the copy of *parm1* on the stack. Hence, procedure *AAMXSND* will receive a *FAR* pointer to the AMX message *parm1, parm2..* in *ES:SI* as required.

Purpose **Add to Bottom of Circular List**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
char      item;                /* one byte item            */
int       status;
 .
 .
status = ajabl(&list, item);
AX             ES:BX  CL,CX or DX:CX
```

Where *&list* is a pointer to a circular list (see *ajrstl*).

item is the 1, 2 or 4 byte item to be added to the list.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

0	=	Added OK; list not full
1	=	Added OK; list now full
-1	=	Cannot add item; list is full

See Also *ajrstl, ajatl, ajrtl, ajrbl*

Purpose **Add to Top of Circular List**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
char      item;                /* one byte item            */
int       status;
 .
 .
status = ajatl(&list, item);
AX             ES:BX  CL,CX or DX:CX
```

Where *&list* is a pointer to a circular list (see *ajrstl*).

item is the 1, 2 or 4 byte item to be added to the list.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

0	=	Added OK; list not full
1	=	Added OK; list now full
-1	=	Cannot add item; list is full

See Also *ajrstl, ajabl, ajrtl, ajrbl*

Purpose **Add to Buffer's Use Count**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
int       increment;
char      *buffp;
int       status;
 .
 .
status = ajbau(buffp, increment);
AX              ES:BX  DX
```

Where *buffp* is a pointer to a buffer obtained with an *ajbget* call.

increment is the signed value to be added to the buffer use count.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

AEROK	=	Call successful use count = use count + *increment*
AERBNU	=	Buffer not in use
AERBUV	=	Use count + *increment* >= 2^{16}
AERNSP	=	Buffer does not belong to a buffer pool

Once a buffer's use count is increased to *n*, the buffer will not be returned to the free list of its pool until *n* calls to *ajbfre* are made to release the buffer.

See Also *ajbget, ajbfre*

Purpose **Create a Buffer Pool**

Used by ■ Task □ ISP □ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
#include "amx831sd.h"
 .
 .
AMXID     poolid;
struct amxbps pooldef;
int       status;
 .
 .
status = ajbcre(&pooldef, &poolid);
AX              ES:BX     DX=
```

Where *&pooldef* is a pointer to the pool definition which describes the buffer pool to be created.

The pool definition structure *amxbps* is defined in header file *AMX831SD.H* (see Appendix D).

All parameters in the pool definition must be valid before *ajbcre* is called. The Buffer Manager will not modify your pool definition.

pooldef.ambpnb = NPN = number of buffers in the pool

pooldef.ambpbs = SPN = usable size of each buffer in bytes >= 8 *SPN* must be even.

pooldef.ambpbp = a pointer to a word aligned region of *N = (SPN+4)*NPN* bytes of contiguous alterable memory (RAM) for the buffer pool.

pooldef.ambptagi = a 4-character name tag for the buffer pool.

&poolid is a pointer to storage for the buffer pool id of the buffer pool allocated to the caller.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

AEROK	=	Call successful
AERNFP	=	No free buffer pool
AERNBF	=	No buffers defined in your pool definition
AERBTS	=	Buffer size defined in your pool definition is too small

If the call is unsuccessful, *poolid* will be undefined.

See Also *ajbdel, ajbtag*

Purpose **Delete a Buffer Pool**

Used by ■ Task □ ISP □ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
AMXID     poolid;
int       status;
 .
 .
status = ajbdel(poolid);
AX               DX
```

Where *poolid* is the pool id of the buffer pool to be deleted.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

AEROK	=	Call successful
AERNSP	=	Invalid pool id

Restrictions No buffer management operations affecting buffers in the pool or alteration of the content of any of the buffers in the pool may take place during or after deletion of the pool.

Only recommended for applications where it is certain that all buffers in the pool are no longer in use.

See Also *ajbcre*

Purpose **Free a Buffer**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
char      *buffp;
int       status;
 .
 .
status = ajbfre(buffp);
AX               ES:BX
```

Where *buffp* is a pointer to a buffer obtained by an *ajbget* call.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

AEROK	=	Call successful
AERBNU	=	Buffer not in use
AERNSP	=	Buffer does not belong to a buffer pool

The buffer use count is decremented by one. If the use count goes to zero, the buffer is returned to its pool's list of free buffers.

See Also *ajbget, ajbau*

Purpose **Get a Buffer**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
AMXID     poolid;
char      *buffp;
int       status;
 .
 .
status = ajbget(poolid, &buffp);
AX              DX      ES:BX=
```

Where *poolid* is the pool id of the pool from which the buffer is to be obtained.

&buffp is a pointer to storage for the returned pointer to the buffer.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

AEROK	=	Call successful
AERNSP	=	Invalid pool id
AERWNB	=	No free buffer available

If a buffer cannot be obtained, the value of *buffp* will be undefined.

If a buffer is allocated, the buffer use count is set to one.

See Also *ajbfre, ajbau, ajbgsz*

Purpose **Get Size of Buffer**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
char      *buffp;
unsigned int size;
int       status;
.
.
status = ajbgsz(buffp, &size);
AX              ES:BX  CX=
```

Where *buffp* is a pointer to a buffer obtained with an *ajbget* call.

size is the size of the buffer in bytes.

Results Interrupts are untouched.

Status is returned.

AEROK	=	Call successful
AERBNU	=	Buffer not in use
AERNSP	=	Buffer does not belong to a buffer pool

If the buffer is not in use, a *size* of 0 is returned.

See Also *ajbget*

Purpose **Initialize (Reset) All Buffer Pools**

Used by ■ Task □ ISP □ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
int       status;
 .
 .
status = ajbia();
AX
```

Results Interrupts are disabled and then restored to their state at the time of the call.

`Status` is returned.

`AEROK`	=	Call successful
`AERBTS`	=	Buffer size defined for some pool is too small If this error occurs, the Buffer Manager's private data has been corrupted. You should consider this error to be fatal.

Restrictions This procedure would only be called when your AMX system is running if it is your intent to reset ALL of your currently defined buffer pools. It is called automatically by the Buffer Manager when your AMX system is started.

No buffer management operations or alteration of buffer contents may take place during initialization. All buffers in all pools are initialized even if they are currently in use. Contents of buffers will be altered.

See Also `ajbip`

Purpose **Initialize (Reset) One Buffer Pool**

Used by ■ Task □ ISP □ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
AMXID     poolid;
int       status;
 .
 .
status = ajbip(poolid);
AX              DX
```

Where *poolid* is the pool id of the buffer pool to be initialized.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

AEROK	=	Call successful
AERNSP	=	Invalid pool id
AERBTS	=	Buffer size defined for the pool is too small If this error occurs, the Buffer Manager's private data has been corrupted. You should consider this error to be fatal.

Restrictions This procedure should only be used in applications in which it is certain that buffers will not be used or released during the initialization of the buffer pool.

No buffer management operations or alteration of buffer contents for any of the buffers in the pool may take place during initialization. All buffers in the pool are initialized even if they are currently in use. Contents of buffers will be altered.

See Also *ajbia*

Purpose **Find a Buffer Pool**

Used by ■ Task □ ISP □ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
char      tag[4];
AMXID     poolid;
int       status;
 .
 .
status = ajbtag(&poolid, tag);
AX              DX=      [DX:CX] see note
```

Where *&poolid* is a pointer to storage for the pool id of the buffer pool of interest.

tag is a pointer to a 4-character name identifying the buffer pool of interest.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

AEROK	=	Call successful
AERNSP	=	No buffer pool with matching tag can be found

If more than one buffer pool was created with the same tag, you will get back the pool id of one of them, but which one is not certain.

If a buffer pool with the given tag cannot be located, the pool id in *poolid* will be undefined.

Buffer pool tags are 4 characters long. All characters must match your search tag exactly.

Note A tag *'ABCD'* is presented in register *DX:CX* with *'A'* in *CL* and *'D'* in *DH*.

See Also *ajbcre*

Purpose **ajcfjset Sets a Mark for a Long Jump**
ajcfjlong Long Jumps to that Mark

These procedures are provided for AMX portability. They are not replacements for C library procedures *longjmp* or *setjmp* although they function in a similar manner.

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup Prototype is in file *AMX831CF.H*.

```
#include "AMX831CF.H"
void cdecl ajcfjlong(struct ajxjbuf *jbuf, int value);
int cdecl ajcfjset(struct ajxjbuf *jbuf);
```

Where *jbuf* is a pointer to a jump buffer to be used to mark the processor state at the time *ajcfjset* is called and to restore that state when *ajcfjlong* is subsequently called.

The processor dependent structure *ajxjbuf* is defined in file *AMX831SD.H*.

value is an integer value to be returned to the *ajcfjset* caller when *ajcfjlong* initiates the long jump return. *Value* cannot be 0. If *value = 0*, *ajcfjlong* will replace it with *value = 1*.

Returns *Ajcfjset* returns 0 when initially called to establish the mark. *Ajcfjset* returns *value* (non 0) when *ajcfjlong* is called to do the long jump to the mark established by the initial *ajcfjset* call.

There is no return from *ajcfjlong*.

Interrupts are not touched by *ajcfjset* or *ajcfjlong*.

Restrictions *Ajcfjset* must be called prior to any call to *ajcfjlong*. Each call must reference the same jump buffer. The jump buffer must remain unaltered between the initial *ajcfjset* call and the subsequent *ajcfjlong* long jump return.

Under no circumstances should one task attempt a long jump using a jump buffer set by another task.

Example

```
#include "AMX831CF.H"

void cdecl dowork(struct ajxjbuf *jbp);

static struct ajxjbuf jumpbuffer;

#define STACKSIZE 512        /* Stack size (longs)        */
static long newstack[STACKSIZE];

                             /* Top of stack (grows down) */
#define STACKP (&newstack[STACKSIZE - 1])

void cdecl taskbody(void) {

  if (ajcfjset(&jumpbuffer) == 0)

        /* Switch to new stack and do work              */
        ajcfstkjmp(&jumpbuffer, STACKP,
                  (void (*)())dowork);
        /* Never returns to here                        */

  /* Do work using original stack                       */
  dowork(NULL);
  }


void cdecl dowork(struct ajxjbuf *jbp) {

  /* Do work                                            */

  /* If jump buffer provided, then use long jump to     */
  /* restore the original stack and return              */
  if (jbp != NULL)
        ajcfjlong(jbp, 1);
  }
```

See Also *ajcfstkjmp*

Purpose **Switch Stacks and Jump to a New Procedure**

This procedure is provided for AMX portability.

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup Prototype is in file *AMX831CF.H*.

```
#include "AMX831CF.H"
void cdecl ajcfstkjmp(void *vp, void *stackp,
          void (*procp)());
```

Where *vp* is a pointer which is passed as a parameter to the new procedure.

stackp is a pointer to a properly aligned block of memory for use as a stack. For the 80x86 family, the stack must be 16-bit word aligned.

Stackp must point to the top of the memory block since the processor stack builds downward.

procp is a pointer to the new procedure which is prototyped as follows:

```
void cdecl newfunc(void *vp);
```

For portability using different C compilers, cast your procedure pointer as *(void (*)())newfunc* in your call to *ajcfstkjmp*.

Interrupts □ Disabled □ Enabled □ Restored

Returns There is no return from *ajcfstkjmp*. Use *ajcfjset* and *ajcfjlong* if there is a requirement to return to the original stack.

Interrupts are not touched by *ajcfstkjmp*.

Restrictions The new procedure referenced by *procp* must never return. The procedure can call *ajend* to end the calling task.

Example See the example provided with *ajcfjset* and *ajcfjlong*.

See Also *ajcfjlong, ajcfjset, ajend*

Purpose **AMX Clock Handler**

Used by □ Task ■ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure

Setup Interrupts must be disabled.
```
ajclk();
```

Results Interrupts may be enabled and/or disabled and will then be restored to their state at the time of the call.

Note This procedure is for Clock Interrupt Service Procedure use only.

Restrictions If you code your clock ISP in C, it must be coded as a C interrupt procedure as described in Chapter 5.2. The ISP must dismiss the clock interrupt and call `ajclk`.

Interrupts must be disabled upon entry to `ajclk` or `AACLK`.

Purpose **Disable or Enable Interrupts**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup
```
ajdi();
ajei();
```

Results Interrupts are disabled by `ajdi()` or enabled by `ajei()`.

The interrupt enable flag (`IF`) in the processor status register (`FLAGS`) is reset to `0` to disable interrupts or set to `1` to enable interrupts.

Restrictions Interrupts should be enabled within a short time after they are disabled or system performance will be degraded.

ISPs must not enable interrupts unless nested interrupts are supported in your application by hardware such as an Intel 8259 Interrupt Controller.

Note These functions can be called from your `main()` program before AMX has been launched or after AMX has been shut down.

See Also `ajei, ajflagrd, ajflagrddi, ajflagwr`

Purpose **End Execution of a Task**

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure

Setup `ajend();`

Results There is no return from `ajend`.

If any task is waiting for this task to finish processing its message, AMX will automatically call `ajwakc` to wake that task.

Restrictions `Ajend` can only be used to terminate the task which is making the call.

See Also `ajsend, ajtrig, ajwakc, ajtkstp, ajtkill, ajtkdel`

Purpose **Launch (Enter) the AMX Multitasking System**

Used by ☐ Task ☐ ISP ☐ Timer Procedure ☐ Restart Procedure ☐ Exit Procedure

Setup

```
#include "amx831sd.h"
 .
 .
struct amxupts FAR *uptp;
int       launchparm;
char      *result;
int       status;
 .
 .
ajupt(&uptp);
status = ajentr(launchparm, uptp,  &result);
AX              BX          ES:SI  DX:BX=
```

Where *launchparm* is a launch parameter bit mask.

bit *0*	Exit control
= *0*	permanent launch
= *AMLPTMP*	temporary launch
bit *1*	Interrupt Vector Table access
= *0*	IVT cannot be accessed
= *AMLPVA*	entries in IVT can be altered
bit *2*	Interrupt state during launch
= *0*	interrupts disabled
= *AMLPIE*	interrupts enabled

uptp is a *FAR* pointer to your User Parameter Table.

The structure *amxupts* and mnemonics *AMLPxxx* are defined in header file *AMX831SD.H* (see Appendix D).

&result is a pointer to 4 bytes of storage for the exit information returned by your *ajexit* call. If *&result* is *NULL* (*0L*), the parameter given to *ajexit* is not returned to the *ajentr* caller. In this example, *result* will receive a pointer to a character.

Results No return unless a task calls *ajexit*.

status = *errcode* given to *ajexit*
result = 4 byte parameter *exitinfo* given to *ajexit*

Upon return, interrupts are restored to their state upon entry to this procedure.

Used By Must only be called by a C *main()* procedure to start an AMX system.

See Also *ajexit*

Purpose **Create an Event Group**

Used by ■ Task □ ISP □ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
AMXID     group;
unsigned int ivalue;
int       status;
char      tag[4];
 .
 .
status = ajevcre(&group, ivalue, tag);
AX               BX=     BX      [DX:CX]
```

Where *&group* is a pointer to storage for the event group id of the event group allocated to the caller.

ivalue is the initial value of the 16 event flags in the group.

tag is a 4-character name tag for the event group.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

AEROK	=	Call successful
AERNEB	=	No free event group

If the call is unsuccessful, *group* will be undefined.

See Also *ajevdel, ajevtag*

Purpose **Delete an Event Group**

Used by ■ Task □ ISP □ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
AMXID     group;
int       status;
 .
 .
status = ajevdel(group);
AX                BX
```

Where *group* is the group id of an event group acquired with a call to *ajevcre*.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

AEROK	=	Call successful
AEREVU	=	Event group is still in use One or more tasks are still waiting for events in the group.
AERNSG	=	Invalid event group id

Restrictions You must be absolutely certain that no other task, ISP or Timer Procedure is in any way using or about to use the event group. Failure to observe this restriction may lead to unexpected and unpredictable faults.

See Also *ajevcre*

Purpose **Get the Saved Event Flags**

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup

```
unsigned int value;
 .
 .
value = ajevnt();
AX
```

Where *value* is the state of the 16 event flags at the time the calling task most recently completed a call to *ajevwat.*

Results Interrupts are untouched.

A task can wait for events in an event group by calling *ajevwat*. *Ajevnt* is used to retrieve the state of the event flags at the time the *ajevwat* call completed.

Ajevwat saves a copy of the current state of all 16 event flags in the calling task's Task Control Block at the moment the calling task's event match occurs or its wait interval expires. Subsequent calls by the task to *ajevnt* will retrieve these saved event flags.

Ajevnt is paired with a task's most recent *ajevwat* call. *Ajevwat* only updates the saved event flag copy if an event match or timeout occurs. If *ajevwat* has never updated the task's saved event flag copy, then the *value* returned by *ajevnt* will be undefined. *Ajevnt* will continue to return the same value until the task makes another call to *ajevwat* which results in an update of the saved event flag copy.

A task can only retrieve its own saved event flags. A task cannot retrieve the saved event flags of another task.

See Also *ajevwat, ajevrd*

Purpose **Read the Current Event States in an Event Group**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
AMXID    group;
unsigned int value;
int      status;
 .
 .
status = ajevrd(group, &value);
AX              BX     CX=
```

Where *group* is the group id of an event group acquired with a call to *ajevcre.*

&value is a pointer to storage to receive the current state of the 16 event flags of the particular event group.

Results Interrupts are untouched.

Status is returned.

AEROK	=	Call successful
AERNSG	=	Invalid event group id

If status is not *AEROK*, the variable *value* is unaltered.

Note *Ajevrd* differs from *ajevnt*. *Ajevrd* gives you the current state of all 16 event flags of a particular event group. *Ajevnt* gives a task the state of all 16 event flags at the completion of that task's most recent call to *ajevwat*.

See Also *ajevnt*

Purpose **Signal Event(s) in an Event Group**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
AMXID     group;
unsigned int mask;
unsigned int value;
int       status;
 .
 .
status = ajevsig(group, mask, value);
AX               see note
```

Where *group* is the group id of an event group acquired with a call to *ajevcre*.

mask is a 16-bit mask identifying the event flags of interest in the group.

value is a 16-bit value which specifies the desired states for each of the event flags selected by the mask. The states of flags not specified by the mask can be either set or clear.

Results Interrupts are disabled and then restored to their state at the time of the call.

Task switching is disabled and then restored to its state at the time of the call.

Status is returned.

AEROK	=	Call successful
AERNME	=	Cannot signal the event(s) Probably implies that no AMX message envelopes are available for use.
AERNSG	=	Invalid event group id

An immediate task switch will occur if a task which was waiting for the new event state is of higher priority than the current task.

Note Assembly language call is:

```
PUSH        value
PUSH        mask
PUSH        group
CALL        AAEVSIG
ADD         SP,6
```

See Also *ajevwat*

Purpose **Find an Event Group**

Used by ■ Task □ ISP □ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
AMXID     group;
char      tag[4];
int       status;
 .
 .
status = ajevtag(&group, tag);
AX                BX=     [DX:CX] see note
```

Where *&group* is a pointer to storage for the event group id if the event group of interest.

tag is a pointer to a 4-character name identifying the event group of interest.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

AEROK	=	Call successful
AERNSG	=	No event group with matching tag can be found

If more than one event group was created with the same tag, you will get back the event group id of one of them, but which one is not certain.

If an event group with the given tag cannot be located, the event group id in *group* will be undefined.

Event group tags are 4 characters long. All characters must match your search tag exactly.

Note A tag *'ABCD'* is presented in register *DX:CX* with *'A'* in *CL* and *'D'* in *DH*.

See Also *ajevcre*

Purpose **Wait for Event(s) in an Event Group**

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup

```
AMXID     group;
unsigned int mask;
unsigned int value;
int       match;
long      timeout;
int       status;
 .
 .
status = ajevwat(group, mask, value, match, timeout);
AX               see note
```

Where *group* is the group id of an event group acquired with a call to *ajevcre*.

mask is a 16-bit mask identifying the event flags of interest in the group.

value is a 16-bit value which specifies the states of interest for each of the event flags selected by the mask. The states of flags not specified by the mask can be either set or clear.

match defines the event match requirements:

0	Any selected flag in its required state is considered the event of interest.
<> 0	All selected flags must be in the state specified by value to be considered the event of interest.

timeout is the maximum interval measured in AMX system ticks which the caller is prepared to wait for an event match. Timeout must be positive. If *timeout* = 0, the caller will wait forever for the specified event match.

Results Interrupts are disabled and then enabled upon return.

Status is returned.

AEROK	=	Call successful; event match occurred
AERTMO	=	Timed out before event match occurred
AERTMV	=	Invalid timeout interval (<0)
AERNSG	=	Invalid event group id

If the events in the group match your event selection criterion when you call *ajevwat*, the calling task continues execution immediately without waiting.

Upon return from *ajevwat* after a match or a timeout, the state of all 16 event flags at the time of the return are saved in the calling task's Task Control Block. Procedure *ajevnt* can be called to retrieve the saved copy of the event flags. If you waited for any one of several events, you can interpret these flags to determine which of the events caused your task to resume. If you timed out waiting for all of several events to occur, you can interpret these flags to determine which events did not occur.

Note Assembly language call is:

```
PUSH    MS word of timeout
PUSH    LS word of timeout
PUSH    match
PUSH    value
PUSH    mask
PUSH    group
CALL    AAEVWAT
ADD     SP,12
```

See Also *ajevnt, ajevsig*

Purpose **Leave (Exit) the AMX Multitasking System**

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure

Setup

```
char      *exitinfo;
int       errcode;
 .
 .
ajexit(errcode, exitinfo);
       CX       DX:BX
```

Where *errcode* is the error code to be returned to the *ajentr* caller as *status*.

exitinfo is any 4-byte parameter to be copied to the *ajentr* caller's *result* variable. In this example, the pointer value *exitinfo* will be returned to the *ajentr* caller.

Results There is no return from *ajexit*.

Execution resumes in procedure *ajentr*.
errcode is returned to the *ajentr* caller as *status*.
exitinfo is copied to the *ajentr* caller's *result* variable.

Restrictions Your AMX system must be shut down. All task operation and device I/O must be terminated before *ajexit* is called.

If AMX was started for permanent execution (*launchparm[0]* = 0 on entry to *ajentr*), AMX will take its fatal exit via *ajfatl* with error code *AERFX4* if you try to stop AMX.

See Also *ajentr, ajfatl*

> Note
>
> If any Exit Procedure is coded using the Medium model, any task which calls *ajexit* MUST also use the Medium model.

Purpose **Fatal Exit from AMX**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
int       errcode;
 .
 .
ajfatl(errcode);
       CX
```

Where *errcode* is an error code which will be passed to your Fatal Exit Procedure if one was provided in your User Parameter File.

Results There is no return from *ajfatl*.

If you have not provided a Fatal Exit Procedure or if it returns to AMX, AMX will halt at location *AMHALT*.

See Also *ajexit*

Purpose **Read Processor Flags**
Read Processor Flags and Disable Interrupts
Write Processor Flags

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
unsigned int flags;
 .
 .
flags = ajflagrd();          /* Read current flags        */
flags = ajflagrddi();        /* Read current flags and    */
                             /* then inhibit interrupts   */
 .
 .
ajflagwr(flags);             /* Restore flags             */
```

Results *Ajflagrd* and *ajflagrddi* return the state of the processor status register (*FLAGS*) at the time of the call.

Upon return from *ajflagrddi*, interrupts are disabled.

Upon return from *ajflagrd*, interrupts are untouched.

Ajflagwr writes parameter *flags* to the processor status register (*FLAGS*) and returns nothing.

Upon return from *ajflagwr*, interrupts are enabled or disabled according to the state of the interrupt enable flag (*IF*) in parameter *flags*.

Restrictions Interrupts should be enabled within a short time after they are disabled or system performance will be degraded.

Note The assembly language sequence for *ajflagrd* and *ajflagrddi* is:

```
PUSHF
POP         AX                ;AX = processor flags
CLI                           ;Omit for ajflagrd
```

The assembly language sequence for *ajflagwr* is:

```
PUSH        <flags>
POPF
```

Note These functions can be called from your *main()* program before AMX has been launched or after AMX has been shut down.

See Also *ajdi, ajei*

Purpose **Get Message from Task Mailbox**

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup

```
int       priority;
char      msg[AMXMSZ];
int       status;
 .
 .
status = ajgmsg(priority, msg);
AX              CX        ES:BX
```

Where *priority* is the priority of the task mailbox from which the message is to be fetched.
(0 = highest; 3 = lowest)
(4 = highest priority available message)

msg is an array name or pointer to *AMXMSZ* consecutive bytes of storage into which the message will be copied if it is available. The message storage variable may be any *AMXMSZ* byte structure. *AMXMSZ* is configured in the User Parameter File.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

AEROK	=	Call successful
AERCWT	=	Task's caller is still waiting
AERNMG	=	No message waiting in the specified task mailbox

If *status* is not *AEROK*, the variable *msg* is unaltered.

If you get status *AERCWT*, the task which is making the *ajgmsg* call is currently processing a message sent from a task which is waiting for acknowledgement that its message has been received. You cannot get a new message until you finish with the current one. Call *ajwakc* to wake the waiting task and then call *ajgmsg* again.

See Also *ajsend, ajsenw, ajwakc*

Purpose **Get Pointer Offset**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
unsigned int ofs;
char FAR *pntr;
 .
 .
ofs = ajgofs(pntr);
```

Results Interrupts are untouched.

The 16-bit unsigned integer value of the offset part of the *FAR* pointer variable *pntr* is returned in *ofs*.

See Also *ajgseg, ajsofs, ajsseg*

Purpose **Get Pointer Segment**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
unsigned short int seg;
char FAR *pntr;
 .
 .
seg = ajgseg(pntr);
```

Results Interrupts are untouched.

The 16-bit unsigned integer value of the segment selector part of the *FAR* pointer variable *pntr* is returned in *seg*.

See Also *ajgofs, ajsofs, ajsseg*

Purpose **Get Segment Registers**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
#include "amx831sd.h"
 .
 .
struct amxsregs sregarray;
 .
 .
ajgsreg(&sregarray);
```

Where *&sregarray* is a pointer to storage to receive the current contents of the 8086 segment registers. The structure *amxsregs* is defined in header file *AMX831SD.H* (see Appendix D).

Results Interrupts are untouched.

See Also *ajssreg*

Purpose **Install Task Scheduler Hooks**

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup

```
struct {
          int     (*uhstart)();
          int     (*uhend)();
          int     (*uhsuspend)();
          int     (*uhresume)();
        } userhooks;
 .
 .
ajhook((void FAR *)&userhooks);     /* Install hooks      */
                   ES:BX
 .
 .
ajhook(NULL);                       /* Remove hooks       */
       ES:BX=0
```

Where *&userhooks* is a *FAR* pointer to an array of four pointers to your scheduler procedures.

Your procedures are called by the AMX Task Scheduler whenever tasks start, end, suspend or resume execution.

To inhibit the AMX Task Scheduler from any further calls to your procedures, call *ajhook* with a *NULL (0L)* pointer having a null selector as illustrated.

Results Interrupts are untouched.

Purpose **Read an 8-Bit Input Port (Byte)**
Read a 16-Bit Input Port (Word)

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
int       portno;
char      portbyte;
short int portword;
 .
 .
portbyte  = ajinb(portno);
portword  = ajinw(portno);
```

Where *portno* is the port number.

portbyte is the data byte read.

portword is the data word read.

Results Interrupts are untouched.

See Also *ajoutb, ajoutw*

Purpose **Begin Interrupt Service**

Used by □ Task ■ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure

Setup `ajint();`

Results Interrupts are disabled.

The AMX Interrupt Supervisor saves all registers on the caller's stack. If a task has just been interrupted, AMX switches to the AMX Interrupt Stack before returning to the `ajint` caller.

Purpose **End Interrupt Service**

Used by □ Task ■ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure

Setup `ajinx();`

Results Interrupts are disabled.

The AMX Interrupt Supervisor checks to see if the ISP is returning to an interrupted task. If so, it switches back to the task's stack. The saved registers are restored from the stack before returning to the `ajinx` caller.

Restrictions Upon entry to `ajinx`, the stack must be as it was upon return from procedure `ajint`.

Purpose **Make a Conforming ISP Root**

This procedure creates an AMX root ISP permitting a standard C function to serve as an AMX Interrupt Service Procedure.

Used by ■ Task □ ISP □ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
#include "amx831sd.h"
 .
 .
struct amxisps isproot;
void     intproc();
 .
 .
ajispm(intproc, &isproot);
       CX:DX    ES:BX
```

Where *intproc* is a pointer to a C procedure to be called from the root ISP.

&isproot is a pointer to storage in which a conforming AMX root ISP can be constructed. The structure *amxisps* is defined in header file *AMX831SD.H* (see Appendix D).

Results Interrupts are untouched.

The pointer to the root ISP, *&isproot*, can be installed in the Interrupt Vector Table using *ajivtw* or *ajivtx*.

Restrictions Procedure *intproc* is NOT an interrupt procedure. It is a standard C function which is called from the ISP created in *isproot*. Therefore, *intproc* must NOT call *AAINT* or *AAINX* and must not end with an *IRET* instruction.

See Also *ajivtw, ajivtx*

Purpose **Install a Task Trap Handler**

Install a task trap handler to service divide error, overflow or bound check error traps.

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure □ Exit Procedure

Setup

```
int       inttype;
void      handler();
int       status;
 .
 .
status = ajitrp(inttype, handler);
AX              DX       ES:BX
```

Where *inttype* is the 8086 interrupt type

0 = Divide by 0 trap
4 = *INTO* overflow trap
5 = *BOUND* bounds trap

handler is a pointer to the task's trap handler for the particular error trap.

Results Interrupts are untouched.

Status is returned.

AEROK	=	Call successful
AERITT	=	Interrupt type is not one for which task traps are allowed.

Purpose **Read an Interrupt Vector**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
int       inttype;
void      (*oldproc)();
int       status;
 .
 .
status = ajivtr(inttype, &oldproc);
AX               DX       ES:BX
```

Where *inttype* is the 8086 interrupt type (0-255).

&oldproc is a pointer to storage for a copy of the function pointer from the specified entry in the Interrupt Vector Table.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

AEROK	=	Call successful
AERIPR	=	Vector table is not accessible AMX was launched with access to the Interrupt Vector Table denied.

See Also *ajivtw, ajivtx*

Purpose **Write an Interrupt Vector**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
int       inttype;
void      newproc();
int       status;
 .
 .
status = ajivtw(inttype, newproc);
AX               DX       ES:BX
```

Where *inttype* is the 8086 interrupt type (0-255).

newproc is a pointer to the new interrupt handler.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

AEROK	=	Call successful
AERIPR	=	Vector table is not accessible AMX was launched with access to the Interrupt Vector Table denied.

Restrictions You must NOT use this procedure to alter the vector entries for the divide error, overflow trap or bounds error. Use *ajitrp* for that purpose.

See Also *ajitrp, ajivtr, ajivtx*

Purpose **Exchange an Interrupt Vector**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
int       inttype;
void      (*oldproc)();
void      newproc();
int       status;
 .
 .
status = ajivtx(inttype, newproc, &oldproc);
AX               DX       ES:BX    DS:DI
```

Where *inttype* is the 8086 interrupt type (0-255).

newproc is a pointer to the new interrupt handler.

&oldproc is a pointer to storage for the pointer to the current interrupt handler.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

AEROK	=	Call successful
AERIPR	=	Vector table is not accessible AMX was launched with access to the Interrupt Vector Table denied.

Note *oldproc* can be used as *newproc* in a subsequent call to *ajivtx*.

```
status = ajivtx(inttype, oldproc, &oldproc);
```

Restrictions You must NOT use this procedure to alter the vector entries for the divide error, overflow trap or bounds error. Use *ajitrp* for that purpose.

See Also *ajitrp, ajivtr, ajivtw*

Purpose **Create an Empty List**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
#include "amx831sd.h"
 .
 .
struct amxlhs list;
int       offset;
 .
 .
ajlcre(&list, offset);
       DS:SI  CX
```

Where *&list* is a pointer to the list header.

offset is the node offset (in bytes) at which the list node is located in application objects to be linked in this list.

Results Interrupts are untouched.

The list header is initialized to define an empty list.

Purpose **Find First Object on List**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
#include "amx831sd.h"
 .
 .
struct amxlhs list;
struct appobj *object;
 .
 .
object = ajlhead(&list);
ES:BX             DS:SI
```

Where *&list* is a pointer to the list header.

Results Interrupts are untouched.

object = pointer to the first object on the list.

object = *NULL* if the list is empty.

The object remains on the list.

Purpose **Insert Object before Current Object on List**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
#include "amx831sd.h"
 .
 .
struct amxlhs list;
struct appobj *newobj, *curobj;
 .
 .
ajlinsc(&list, newobj, curobj);
        DS:SI  ES:BX   CX:DI
```

Where *&list* is a pointer to the list header.

newobj is a pointer to the new object to be inserted before the object specified by *curobj.*

curobj is a pointer to a particular object on the list.

Results Interrupts are disabled and then restored to their state at the time of the call.

Purpose **Insert Object at Head of List**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
#include "amx831sd.h"
 .
 .
struct amxlhs list;
struct appobj *object;
 .
 .
ajlinsh(&list, object);
        DS:SI  ES:BX
```

Where *&list* is a pointer to the list header.

object is a pointer to the object to be inserted as the new head of the list.

Results Interrupts are disabled and then restored to their state at the time of the call.

Purpose **Insert Object into Keyed List**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
#include "amx831sd.h"
 .
 .
struct amxlhs list;
struct appobj *object;
unsigned int key;
 .
 .
ajlinsk(&list, object, key);
        DS:SI  ES:BX   CX
```

Where *&list* is a pointer to the list header.

object is a pointer to the object to be inserted into the list.

key is the insertion key. Objects are inserted in order of ascending key values. The object with the smallest key will be at the head of the list. If other objects with the same key value already reside on the list, the new object will be inserted after all other objects with that key.

Results Interrupts are disabled and then restored to their state at the time of the list.

Restrictions The caller must own the list. The list must not be manipulated by other tasks, ISPs or Timer Procedures while this call is in progress. You can use the Semaphore Manager to control ownership of the list if necessary.

ISPs should avoid the use of this procedure unless dealing with very short lists. Use of this procedure with long lists may cause unacceptable timing effects in ISPs.

Purpose **Insert Object at Tail of List**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
#include "amx831sd.h"
 .
 .
struct amxlhs list;
struct appobj *object;
 .
 .
ajlinst(&list, object);
        DS:SI  ES:BX
```

Where *&list* is a pointer to the list header.

object is a pointer to the object to be inserted as the new tail of the list.

Results Interrupts are disabled and then restored to their state at the time of the call.

Purpose **Merge Two Lists**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
#include "amx831sd.h"
 .
 .
struct amxlhs destlist, srclist;
struct appobj *destobj, *srcobj;
 .
 .
ajlmerg(&destlist, &srclist, destobj, srcobj);
        see note
```

Where *&destlist* is a pointer to the destination list header.

&srclist is a pointer to the source list header.

destobj is a pointer to the insertion point on the destination list.

srcobj is a pointer to the extraction point on the source list.

Results Interrupts are disabled and then restored to their state at the time of the call.

All of the objects on the source list are inserted on the destination list before *destobj.* Objects are removed from the source list starting with *srcobj*, wrapping around from the tail to the head, and ending with the object immediately previous to *srcobj.* This list of source objects is then inserted in order before *destobj*. The following example illustrates this process.

Before

```
              srcobj
srclist --- X -- Y -- Z
destlist -- A -- B -- C -- D -- E
                    destobj
```

After

```
srclist --- empty
destlist -- A -- B -- Y -- Z -- X -- C -- D -- E
```

Note Assembly language call is:

```
PUSH  <segment   srcobj>
PUSH  <offset    srcobj>
PUSH  <segment   destobj>
PUSH  <offset    destobj>
PUSH  <segment   scrlist>
PUSH  <offset    scrlist>
PUSH  <segment   destlist>
PUSH  <offset    destlist>
CALL  AALMERG
ADD   SP,16
```

Restrictions You must not merge two lists if either of the lists is a keyed list.

Purpose **Find Next Object on List**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
#include "amx83lsd.h"
 .
 .
struct amxlhs list;
struct appobj *object, *curobj;
 .
 .
object = ajlnext(&list, curobj);
ES:BX            DS:SI  ES:BX
```

Where *&list* is a pointer to the list header.

curobj is a pointer to an object on this list.

Results Interrupts are untouched.

object = pointer to the next object on the list following the object specified by *curobj*.

object = *NULL* if *curobj* is at the tail of the list.

The object remains on the list.

Purpose **Reorder an Object in a Keyed List**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
#include "amx831sd.h"
 .
 .
struct amxlhs list;
struct appobj *object;
unsigned int key;
 .
 .
ajlordk(&list, object, key);
        DS:SI  ES:BX   CX
```

Where *&list* is a pointer to the list header.

object is a pointer to the object on the list which is to be moved within the list.

key is the value of the new key for *object.*

Results Interrupts are disabled and then restored to their state at the time of the call.

Restrictions The caller must own the list. The list must not be manipulated by other tasks, ISPs or Timer Procedures while this call is in progress. You can use the Semaphore Manager to control ownership of the list if necessary.

ISPs should avoid the use of this procedure unless dealing with very short lists. Use of this procedure with long lists may cause unacceptable timing effects in ISPs.

See Also *ajlinsk*

Purpose **Find Previous Object on List**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
#include "amx831sd.h"
 .
 .
struct amxlhs list;
struct appobj *object, *curobj;
 .
 .
object = ajlprev(&list, curobj);
ES:BX            DS:SI  ES:BX
```

Where *&list* is a pointer to the list header.

curobj is a pointer to an object on this list.

Results Interrupts are untouched.

object = pointer to the object on the list immediately preceding the object specified by *curobj.*

object = *NULL* if *curobj* is at head of the list.

The object remains on the list.

Purpose **Remove Object from List**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
#include "amx831sd.h"
 .
 .
struct amxlhs list;
struct appobj *object;
 .
 .
ajlrmvc(&list, object);
        DS:SI  ES:BX
```

Where *&list* is a pointer to the list header.

object is a pointer to the object to be removed from the list.

Results Interrupts are disabled and then restored to their state at the time of call.

Purpose **Remove Object from Head of List**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
#include "amx831sd.h"
 .
 .
struct amxlhs list;
struct appobj *object;
 .
 .
object = ajlrmvh(&list);
ES:BX             DS:SI
```

Where *&list* is a pointer to the list header.

Results Interrupts are disabled and then restored to their state at the time of the call.

object = pointer to the object removed from the head of the list.

object = *NULL* if the list was empty.

Purpose **Remove Object from Tail of List**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
#include "amx831sd.h"
 .
 .
struct amxlhs list;
struct appobj *object;
 .
 .
object = ajlrmvt(&list);
ES:BX             DS:SI
```

Where *&list* is a pointer to the list header.

Results Interrupts are disabled and then restored to their state at the time of the call.

object = pointer to the object removed from the tail of the list.

object = *NULL* if the list was empty.

Purpose **Find Last Object on List**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
#include "amx831sd.h"
 .
 .
struct amxlhs list;
struct appobj *object;
 .
 .
object = ajltail(&list);
ES:BX             DS:SI
```

Where *&list* is a pointer to the list header.

Results Interrupts are untouched.

object = pointer to the last object on the list.

object = *NULL* if the list is empty.

The object remains on the list.

Purpose **Add to Memory Block Use Count**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
char      *blockp;
int       increment;
int       status;
 .
 .
status = ajmau(blockp, increment);
AX              ES:BX   DX
```

Where *blockp* is a pointer to a memory block allocated by *ajmget* or *ajmgeh*.

increment is the signed value to be added to the memory block's use count.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

AEROK	=	Call successful use count = use count + *increment*
AERMIB	=	Invalid memory block pointer
AERMNU	=	Memory block not in use
AERMOV	=	Memory block use count overflow

If the call is not successful, the memory block use count is unaltered.

Once a memory block's use count is increased to *n*, the memory will not be freed until *n* calls to *ajmfre* are made to release the memory block.

See Also *ajmget, ajmgeh, ajmfre*

Purpose **Free Previously Allocated Block**

Used by ■ Task □ ISP □ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
char      *blockp;
int       status;
 .
 .
status = ajmfre(blockp);
AX                ES:BX
```

Where *blockp* is a pointer to a memory block allocated by *ajmget* or *ajmgeh*.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

AEROK	=	Call successful
AERMIB	=	Invalid memory block pointer
AERMNU	=	Memory block not in use

The memory block use count is decremented by one. If the use count goes to zero, the memory block is released for reuse.

See Also *ajmget, ajmgeh, ajmau*

Purpose **Get a Memory Block Using a Memory Handle**

Used by ■ Task □ ISP □ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
long      size;
char      *blockp;
long      memsize;
char      *handle;
int       status;
 .
 .
status = ajmgeh(size, &blockp, &memsize, handle);
AX              DX:CX ES:BX=   DX:CX=    ES:BX
```

Where *size* is the number of bytes of memory required.

&blockp is a pointer to storage for the returned pointer to the memory block.

&memsize is a pointer to storage for the actual usable size in bytes of the memory block at *blockp*. *memsize* may be slightly larger than *size*. It is valid to replace *&memsize* with *&size* to update your *size* variable.

handle is a memory handle assigned by the Memory Manager in a previous call to *ajmhan*. Note that the handle is a pointer.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

AEROK = Call successful
blockp is a pointer to a block of memory.
memsize is the actual size of that block.

AERMNA = Memory not available
blockp is undefined.
memsize is the size of the largest block of memory which is currently available using the given handle.

AERMIB = *handle* is an invalid memory handle.
blockp and *memsize* are undefined.

If memory is allocated, the use count for the memory block is set to one.

See Also *ajmau, ajmfre, ajmhan*

Purpose **Get a Memory Block**

Used by ■ Task □ ISP □ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
long      size;
char      *blockp;
long      memsize;
int       status;
 .
 .
status = ajmget(size, &blockp, &memsize);
AX              DX:CX ES:BX=   DX:CX=
```

Where *size* is the number of bytes of memory required.

&blockp is a pointer to storage for the returned pointer to the memory block.

&memsize is a pointer to storage for the actual usable size in bytes of the memory block at *blockp*. *memsize* may be slightly larger than *size*. It is valid to replace *&memsize* with *&size* to update your *size* variable.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

AEROK	=	Call successful *blockp* is a pointer to a block of memory. *memsize* is the actual size of that block.
AERMNA	=	Memory not available *blockp* is undefined. *memsize* is the size of the largest currently available block of memory.

If memory is allocated, the use count for the memory block is set to one.

See Also *ajmau, ajmfre*

Purpose **Get Size of Memory Block**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
char      *blockp;
long      blksize;
int       status;
 .
 .
status = ajmgsz(blockp, &blksize);
AX               ES:BX   DX:CX=
```

Where *blockp* is a pointer to a memory block allocated by *ajmget* or *ajmgeh*.

&blksize is a pointer to storage for the usable size in bytes of the memory block at *blockp*.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

AEROK	=	Call successful *blksize* is the size of the block in bytes.
AERMIB	=	Invalid memory block pointer
AERMNU	=	Memory block not in use

If the call is not successful, the value of *blksize* is undefined.

See Also *ajmget, ajmgeh*

Purpose **Create Memory Handle for Private Allocation**

Used by ■ Task □ ISP □ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
char      *blockp;
long      memsize;
char      *handle;
int       status;
 .
 .
status = ajmhan(blockp, memsize, &handle);
AX              ES:BX   DX:CX    ES:BX=
```

Where

blockp is a pointer to an area of memory whose access is to be controlled by the Memory Manager. Blocks allocated by *ajmget* or *ajmgeh* are suitable.

memsize is the size of the memory block at *blockp*.

&handle is a pointer to storage for a memory handle which must be used for subsequent access to smaller blocks within the larger block. Note that the handle which is returned is a pointer.

Results Interrupts are untouched.

Status is returned.

AEROK	=	Call successful *handle* can be used for access to the block of memory using *ajmgeh*.
AERMNA	=	Memory size is too small

If the handle cannot be created, the value of *handle* is undefined.

See Also *ajmgeh, ajmfre, ajmau*

Purpose **Get the DGROUP Segment Value**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
ajmodl();
```

Results Interrupts are untouched.

The data segment register `DS` is set to the base segment of group `DGROUP`.

This procedure is provided for use with mixed memory models.

Purpose **Set (Fill) Memory**

Used by ■ Task □ ISP □ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
char      *mempntr;
long      memsize;
unsigned short int pattern;
int       status;
 .
 .
status = ajmset(mempntr, memsize, pattern);
AX              ES:BX    DX:CX    SI
```

Where *mempntr* is a pointer to the memory area which is to be filled with the pattern.

memsize is the size in bytes of the memory area at *mempntr*.

pattern is the 16-bit fill pattern.

Results Interrupts are untouched.

Status is returned.

AEROK	=	Call successful
AERMMV	=	Memory overflow *Memsize* is too large. An attempt to fill the memory region would exceed the 1 Mb addressing range.

This procedure is optimized for speed in filling large regions of memory.

The procedure will fill any even or odd sized region of byte or word aligned memory. The least significant 8 bits of *pattern* are stored at **mempntr*. The most significant 8 bits of *pattern* are stored at **(mempntr+1)* provided *memsize* is greater than one. The pattern then repeats up to the extent indicated by *memsize*.

Purpose **Create a Message Exchange**

Used by ■ Task □ ISP □ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
AMXID     exchange;
int       mb0, mb1, mb2, mb3;
char      tag[4];
int       status;
 .
 .
status = ajmxcre(&exchange, mb0, mb1, mb2, mb3, tag);
AX               BX=        @ES:BX (see note)   [DX:CX]
```

Where *&exchange* is a pointer to storage for the message exchange id of the message exchange allocated to the caller.

mb0, mb1, mb2, mb3 are integers defining the maximum number of message envelopes which can reside in each of the message exchange message queues (0 <= *mbi* <= 32767).

tag is the 4-character tag for the message exchange.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

AEROK	=	Call successful. *exchange* contains a valid message exchange id.
AERNXB	=	No free message exchange
AERMBZ	=	Invalid message queue (mailbox) depth

If the call is unsuccessful, *exchange* is undefined.

Note For assembly language users, *ES:BX* contains a pointer to four 16-bit integers defining the maximum depth of each of the four message queues.

See Also *ajmxdel, ajmxtag*

Purpose **Delete a Message Exchange**

Used by ■ Task □ ISP □ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
AMXID     exchange;
int       status;
 .
 .
status = ajmxdel(exchange);
AX                BX
```

Where *exchange* is the exchange id of the message exchange to be deleted.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

AEROK	=	Call successful
AERNSX	=	Invalid message exchange id
AERXIU	=	Message exchange is still in use Messages may still be present in the message exchange or one or more tasks may be waiting for a message to arrive at the message exchange.

Restrictions You must be absolutely certain that no other task, ISP or Timer Procedure is in any way using or about to use the message exchange. Failure to observe this restriction may lead to unexpected and unpredictable faults.

See Also *ajmxcre*

Purpose **Get a Message from a Message Exchange (no wait)**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
AMXID     exchange;
char      msg[AMXMSZ];
int       status;
 .
 .
status = ajmxget(exchange, msg);
AX               BX        ES:SI
```

Where *exchange* is the message exchange id acquired by a call to *ajmxcre*.

msg is an array name or pointer to *AMXMSZ* consecutive bytes of storage into which the message will be copied if it is available. The message storage variable may be any *AMXMSZ* byte structure. *AMXMSZ* is configured in the User Parameter File.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

AEROK	=	Call successful
AERNSX	=	Invalid message exchange id
AERNMG	=	No message available

Note This procedure returns immediately with an error indication if no message is available. Tasks can wait for a message by calling *ajmxwat*.

See Also *ajmxwat, ajmxsnd*

Purpose **Send Message to Message Exchange**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
AMXID     exchange;
char      msg[AMXMSZ];
int       priority;
int       status;
 .
 .
status = ajmxsnd(exchange, priority, msg);
AX               BX        DX        ES:SI
```

Where *exchange* is the message exchange id acquired by a call to *ajmxcre.*

priority is the priority of the message.
(0 = highest, 3 = lowest)

msg is an array name or pointer to *AMXMSZ* consecutive bytes that form the message to be copied to the message exchange. The message may be any *AMXMSZ* byte structure. *AMXMSZ* is configured in the User Parameter File. If *msg* does not reside in the data segment, use procedure *ajmxsndf* with a *FAR* pointer to the *msg*.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

AEROK	=	Call successful
AERNSX	=	Invalid message exchange id
AERNME	=	No free message envelope
AERMBF	=	Message queue (mailbox) is full
AERNMB	=	No message queue (mailbox) of the specified priority

If one or more tasks are waiting at the exchange for a message, the message will be given immediately to the task waiting at the head of the exchange's wait queue. Task rescheduling occurs immediately if necessary.

See Also *ajmxsndp, ajmxwat, ajmxget*

Purpose **Send Message to a Message Exchange**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
AMXID     exchange;
int       priority;
int       status;
int       parm1;
int       parm2;
 .
 .
status = ajmxsndp(exchange, priority, parm1, parm2...);
AX       AAMXSND  BX        DX        @ES:SI
```

Where *exchange* is the message exchange id acquired by a call to *ajmxcre.*

priority is the priority of the message.
(0 = highest, 3 = lowest)

parm1, parm2... on the stack form the message to be copied to the message exchange. The number of bytes copied from the stack depends on the AMX message size which is *AMXMSZ* bytes. *AMXMSZ* is configured in the User Parameter File.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

AEROK	=	Call successful
AERNSX	=	Invalid message exchange id
AERNME	=	No free message envelope
AERMBF	=	Message queue (mailbox) is full
AERNMB	=	No message queue (mailbox) of the specified priority

If one or more tasks are waiting at the exchange for a message, the message will be given immediately to the task waiting at the head of the exchange's wait queue. Task rescheduling occurs immediately if necessary.

Note The equivalent assembly language procedure is *AAMXSND* which receives the parameters shown above.

See Also *ajmxsnd, ajmxwat, ajmxget*

Purpose **Find a Message Exchange**

Used by ■ Task □ ISP □ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
AMXID     exchange;
char      tag[4];
int       status;
 .
 .
status = ajmxtag(&exchange, tag);
AX               BX=        [DX:CX] see note
```

Where *&exchange* is a pointer to storage for the message exchange id of the message exchange of interest.

tag is a pointer to a 4-character name identifying the message exchange of interest.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

AEROK	=	Call successful
AERNSX	=	No message exchange with matching tag can be found

If more than one message exchange was created with the same tag, you will get back the message exchange id of one of them, but which one is not certain.

If a message exchange with the given tag cannot be located, the message exchange id in *exchange* will be undefined.

Message exchange tags are 4 characters long. All characters must match your search tag exactly.

Note A tag *'ABCD'* is presented in register *DX:CX* with *'A'* in *CL* and *'D'* in *DH*.

See Also *ajmxcre*

Purpose **Wait for a Message from a Message Exchange**

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup

```
AMXID     exchange;
char      msg[AMXMSZ];
long      timeout;
unsigned int priority;
 .
 .
status = ajmxwat(exchange, msg,   timeout, priority);
AX               BX        ES:SI  DX:CX    DI
```

Where *exchange* is the message exchange id acquired by a call to *ajmxcre.*

msg is an array name or pointer to *AMXMSZ* consecutive bytes of storage into which the message will be copied if it is available. The message storage variable may be any *AMXMSZ* byte structure. *AMXMSZ* is configured in the User Parameter File.

timeout is the maximum interval measured in AMX system ticks which the caller is prepared to wait for a message. *Timeout* must be positive. If *timeout* = 0, the caller will wait forever for a message to arrive.

priority is the priority at which the caller wishes to wait (0 = highest). To wait in FIFO order, have all callers use the same value for *priority*.

Results Interrupts are disabled and enabled and then restored to their state at the time of the call.

Status is returned.

AEROK	=	Call successful
AERNSX	=	Invalid message exchange id
AERTMO	=	Timed out before message available
AERTMV	=	Invalid timeout interval (<0)

See Also *ajmxget, ajmxsnd*

Purpose **Write to an 8-Bit Output Port (Byte)**
Write to a 16-Bit Output Port (Word)

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
int       portno;
char      portbyte;
short int portword;
 .
 .
ajoutb(portno, portbyte);
ajoutw(portno, portword);
```

Where *portno* is the port number.

portbyte is the data byte to be written.

portword is the data word to be written.

Results Interrupts are untouched.

See Also *ajinb, ajinw*

Purpose **Call Software Procedure**

You can use this procedure to call any software procedure which, because of register setup requirements, cannot otherwise be called from C.

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
#include "amx831sd.h"
 .
 .
void      someproc();
struct amxregs regarray;
unsigned int regeax:
 .
 .
regeax = ajproc(someproc, &regarray);
```

or

```
regeax = ajprocq(someproc, &regarray);
```

Where *someproc* is the name of the *FAR* procedure which you wish to call. The procedure must end with a *FAR* return.

regarray is a structure containing the processor register values which are required for the particular software procedure being called. *Ajproc* requires that all of the segment registers in *regarray* be initialized with valid selectors. *Ajprocq* unconditionally copies the contents of the current hardware segment registers into the segment registers in *regarray*. General register values which are not required by the procedure do not have to be provided.

If registers *DS* and/or *ES* are not required by procedure *someproc*, set their values to 0 in *regarray*.

The structure *amxregs* is defined in header file *AMX831SD.H* (see Appendix D).

regeax is the value which was in register *AX* when the called procedure ended.

Results Interrupts will be in the state in which they are left by the called procedure when it exits.

All registers in *regarray* reflect the values in the processor registers at the time the called procedure ended.

The real processor registers are only affected at the instant the software procedure call is made.

Restrictions The flags register in *regarray* can be set prior to calling *ajproc*. Only the least significant 8 bits of the real flags register are altered to match the value specified in *regarray*.

Upon return, all 16 bits of the flags register in *regarray* are valid.

See Also *ajsint*

Purpose **Lower Task Privilege**

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup *ajprvl();*

Results Interrupts are disabled and then restored to their state at the time of the call.

Restrictions Must follow every call to *ajprvr*. Once *ajprvr* is called, task switching is inhibited until *ajprvl* is called.

See Also *ajprvr*

Purpose **Raise Task Privilege**

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup *ajprvr();*

Results Interrupts are disabled and then restored to their state at the time of the call.

Restrictions Must be followed by a call to *ajprvl* as soon as possible to lower the privilege level.

Task switching is inhibited until *ajprvl* is called.

Once *ajprvr* has been called, the task must not issue any AMX calls which would force the task to wait or end. The task must remain compute bound until it calls *ajprvl* to lower the task's privilege.

See Also *ajprvl*

Purpose **Remove from Bottom of Circular List**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
char      item;                /* one byte item            */
int       status;
 .
 .
status = ajrbl(&list, &item);
AX              ES:BX  CL=, CX= or DX:CX=
```

Where *&list* is a pointer to a circular list (see *ajrstl*).

&item is a pointer to storage for the 1, 2 or 4 byte item to be removed from the list. If *&item* is *NULL* (*0L*), the item is deleted from the list but is not returned to the caller.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned..

0	=	Removed OK; list not empty
1	=	Removed OK; list now empty
-1	=	Cannot remove item; list is empty

See Also *ajrstl, ajatl, ajabl, ajrtl*

Purpose **Resume a Suspended Task**

Resume a task known to be suspended as a result of an *ajsusp* call.

Used by ■ Task ■ ISP ■ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup

```
AMXID     taskid;
int       status;
 .
 .
status = ajresum(taskid);
AX                DX
```

Where *taskid* is the task id of the task to be resumed.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.
AEROK = Call successful
AERNST = Invalid task id

An immediate task switch will occur if the task being resumed is of higher priority than the current task.

See Also *ajsusp*

Purpose **Initialize (Reset) a Circular List**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
/* define type of slots                                    */
typedef char SLOT1;           /* byte slot                 */
typedef short int SLOT2;      /* word slot                 */
typedef long SLOT4;           /* double word slot          */

/* define number of slots in list                          */

#define   NSLOT     64

struct {
          int       header[4];
          SLOT4     slots[NSLOT];
          } list;
 .
 .
ajrstl(&list, sizeof(SLOT4), NSLOT);
       ES:BX  DX             CX
```

Where *&list* is a pointer to storage for use as a circular list.

sizeof(SLOT4) is the slot size of the circular list (1, 2, or 4 bytes corresponding to byte, word or double word slots).

NSLOT is the number of slots in the list. (0 < *NSLOT* < 16380)

Results Interrupts are disabled and then restored to their state at the time of the call.

See Also *ajatl, ajabl, ajrtl, ajrbl*

Purpose **Remove from Top of Circular List**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
char      item;                /* one byte item            */
int       status;
 .
 .
status = ajrtl(&list, &item);
AX             BX     CL=, CX= or DX:CX=
```

Where *&list* is a pointer to a circular list (see *ajrstl*).

&item is a pointer to storage for the 1, 2 or 4 byte item to be removed from the list. If *&item* is *NULL* (*0L*), the item is deleted from the list but is not returned to the caller.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

0	=	Removed OK; list not empty
1	=	Removed OK; list now empty
-1	=	Cannot remove item; list is empty

See Also *ajrstl, ajatl, ajabl, ajrbl*

Purpose **Send a Message to a Task Mailbox**

To request AMX to send a message to a task at a given priority and to begin execution of that task as soon as possible.

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
AMXID     taskid;
int       priority;
char      msg[AMXMSZ];
int       status;
 .
 .
status = ajsend(taskid, priority, msg);
AX              DX      CX        ES:BX
```

Where *taskid* is the task id of the task to whom the message is to be sent.

priority is the priority of the message.
(0 = highest; 3 = lowest).

msg is an array name or pointer to *AMXMSZ* consecutive bytes that form the message to be copied to the destination task. The message may be any *AMXMSZ* byte structure. *AMXMSZ* is configured in the User Parameter File.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

AEROK	=	Call successful
AERNST	=	Invalid task id
AERNME	=	No free message envelope
AERNMB	=	No task mailbox of the specified priority
AERMBF	=	Task mailbox is full

An immediate task switch will occur if the task to whom the message is being sent is idle and is of higher priority than the current task.

See Also *ajsendp, ajsenw*

Purpose **Send a Message to a Task Mailbox**

To request AMX to send a message to a task at a given priority and to begin execution of that task as soon as possible.

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
AMXID     taskid;
int       priority;
int       status;
int       parm1;
int       parm2;
 .
 .
status = ajsendp(taskid, priority, parm1, parm2...);
AX       AASEND  DX      CX        @ES:BX
```

Where *taskid* is the task id of the task to whom the message is to be sent.

priority is the priority of the message.
(0 = highest; 3 = lowest).

parm1, parm2... on the stack form the message to be copied to the destination task. The number of bytes copied from the stack depends on the AMX message size which is *AMXMSZ* bytes. *AMXMSZ* is configured in the User Parameter File.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

AEROK	=	Call successful
AERNST	=	Invalid task id
AERNME	=	No free message envelope
AERNMB	=	No task mailbox of the specified priority
AERMBF	=	Task mailbox is full

An immediate task switch will occur if the task to whom the message is being sent is idle and is of higher priority than the current task.

Note The equivalent assembly language procedure is *AASEND* which receives the parameters shown above.

See Also *ajsend, ajsenw*

Purpose **Send a Message to a Task Mailbox (Wait for Ack)**

To request AMX to send a message to a task at a given priority and to begin execution of that task as soon as possible. The task making the request is placed into the wait state until either:

1. the called task receives the message and makes an *ajwakc* call, or

2. the called task receives the message and ends by making an *ajend* call or by returning to AMX.

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup

```
AMXID     taskid;
int       priority;
char      msg[AMXMSZ];
int       status;
 .
 .
status = ajsenw(taskid, priority, msg);
AX       AASEND DX      CX[14..0] ES:BX
                        CX[15]=1
```

Where *taskid* is the task id of the task to whom the message is to be sent.

priority is the priority of the message.
(0 = highest; 3 = lowest).

msg is an array name or pointer to *AMXMSZ* consecutive bytes that form the message to be copied to the destination task. The message may be any *AMXMSZ* byte structure. *AMXMSZ* is configured in the User Parameter File.

Results Interrupts are disabled and then restored to their state at the time of the call.

`Status` is returned.

`AEROK`	=	Call successful
`AERNST`	=	Invalid task id
`AERNME`	=	No free message envelope
`AERNMB`	=	No task mailbox of the specified priority
`AERMBF`	=	Task mailbox is full

An immediate switch to the highest priority ready task will occur.

If the call is successful, `status` will be >= 0. By definition, the value 0 is the AMX error code `AEROK`. If `status` > 0, it implies that the task to whom the message was sent wakened the caller using `ajwakcs` to return a result to the caller. The positive value in variable `status` is the response returned by that task.

Note The equivalent assembly language procedure is `AASEND` which receives the parameters shown above.

See Also `ajsenwp, ajsend, ajwakc, ajwakcs`

Purpose **Send a Message to a Task Mailbox (Wait for Ack)**

To request AMX to send a message to a task at a given priority and to begin execution of that task as soon as possible.

The task making the request is placed into the wait state until either:

1. the called task receives the message and makes an *ajwakc* call, or
2. the called task receives the message and ends by making an *ajend* call or by returning to AMX.

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup

```
AMXID     taskid;
int       priority;
int       status;
int       parm1;
int       parm2;
 .
 .
status = ajsenwp(taskid, priority,  parm1, parm2...);
AX       AASEND  DX      CX[14..0]  @ES:BX
                         CX[15]=1
```

Where *taskid* is the task id of the task to whom the message is to be sent.

priority is the priority of the message.
(0 = highest; 3 = lowest).

parm1, parm2... on the stack form the message to be copied to the destination task. The number of bytes copied from the stack depends on the AMX message size which is *AMXMSZ* bytes. *AMXMSZ* is configured in the User Parameter File.

Results Interrupts are disabled and then restored to their state at the time of the call.

`Status` is returned.

`AEROK`	=	Call successful
`AERNST`	=	Invalid task id
`AERNME`	=	No free message envelope
`AERNMB`	=	No task mailbox of the specified priority
`AERMBF`	=	Task mailbox is full

An immediate switch to the highest priority ready task will occur.

If the call is successful, `status` will be >= 0. By definition, the value 0 is the AMX error code `AEROK`. If `status` > 0, it implies that the task to whom the message was sent wakened the caller using `ajwakcs` to return a result to the caller. The positive value in variable `status` is the response returned by that task.

Note The equivalent assembly language procedure is `AASEND` which receives the parameters shown above.

See Also `ajsenw, ajsend, ajwakc, ajwakcs`

Purpose **Signal a Task**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
AMXID     taskid;
unsigned int tsignals;
unsigned long siginfo;
int       status;
 .
 .
status = ajsgnl(taskid, tsignals,  &siginfo);
AX              DX      BX[14..0]  BX:CX=
                        BX[15]=0
```

Where *taskid* is the task id of the task to be signalled.

tsignals is a 15-bit mask identifying the task signals to be sent to the specified task. Only the least significant 15 bits of *tsignals* (bits 0 to 14) are used. The meaning of each of these signal bits is up to you. Set a bit to cause the corresponding task signal. Multiple task signals can be specified in the mask. All other bits must be 0.

&siginfo is a pointer to storage for 32 bits of information about the state of the task's 15 task signals at the instant this *ajsgnl* call completes.

If you are signalling only one task signal or are not interested in the additional status returned in *siginfo*, replace *&siginfo* with *NULL (0L)*.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

AEROK = Call successful
Task was waiting for one or more of these signals.
The task is no longer waiting. (See Note).

AERWAT = Warning: Task was waiting for one or more of these signals. The task is still waiting for other signals. (See Note).

AERWKP = Warning: Task was not waiting for any of these signals. These signals are now pending for the task. No signal overruns occurred.

AERNST = Invalid task id

Results (continued)

AERTNW = Task was not waiting for any of these signals. These signals are now pending for the task. At least one of these signals was already pending implying a signal overrun. Variable *siginfo* can be examined to see which signals caused the overrun.

Siginfo provides additional information concerning the results of the signal. Bits 0 to 14 identify signal overruns. If any of these bits is set, it implies that the task was not waiting for that task signal and that the task signal was already pending.

Bits 16 to 30 of *siginfo* indicate which, if any, of all of the task's 15 task signals remain pending after this *ajsgnl* call. Bits 16 to 30 correspond to task signal bits 0 to 14 respectively.

An immediate task switch will occur if the task being signalled is of higher priority than the current task and its signal wait condition has been satisfied as a result of this call.

Note When signalling multiple task signals at once, it is possible to generate a signal overrun and still get the *AEROK* or *AERWAT* result in *status*.

For example, if you generate task signals A and B and the task is waiting for signal A and signal B is already pending, then *status* will return *AEROK* because the task's signal wait condition has been satisfied. However, an overrun on task signal B did occur and will be indicated in variable *siginfo*.

If *tsignals* only specifies one task signal, then *status* completely defines the result. You can replace *&siginfo* with *NULL* if so desired.

AEROK	Task resumed
AERWKP	Task not waiting; signal now pending
AERTNW	Task not waiting; signal overrun

AMX procedure *ajwake* is a simplified form of procedure *ajsgnl* in which one predefined private AMX task signal is used.

See Also *ajsgwat*, *ajsgrd*, *ajsgres*, *ajwake*

Purpose **Read Pending Task Signals**

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup

```
unsigned int tsignals;
 .
 .
tsignals = ajsgrd();
AX
```

Where *tsignals* is the current state of the calling task's task signals.

Results Interrupts are untouched.

tsignals contains the current value of each of the task's 15 task signals in the least significant 15 bits (bits 0 to 14). All other bits of *tsignals* will be 0.

If a task signal bit in *tsignals* is set, it means that the corresponding task signal has been signalled at least once (by some call to *ajsgnl*) following the most recent call by this task to *ajsgres* to reset that task signal OR to *ajsgwat* to wait for that task signal.

Restriction A task can only read its own task signals. It cannot read the task signals of any other task.

See Also *ajsgwat*, *ajsgnl*, *ajsgres*

Purpose **Reset Pending Task Signals**

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup

```
unsigned int tsignals;
 .
 .
ajsgres(tsignals);
        BX[14..0]
        BX[15]=0
```

Where *tsignals* is a 15-bit mask identifying the task signals to be reset. Only the least significant 15 bits of *tsignals* (bits 0 to 14) are used. Set a bit to force the reset of the corresponding task signal. All other bits must be 0.

Results Interrupts are untouched.

The task signals specified by *tsignals* are reset. All other task signals are unaltered.

Restriction A task can only reset its own task signals. It cannot reset the task signals of any other task.

Note AMX procedure *ajwapr* is a simplified form of procedure *ajsgres* in which one predefined private AMX task signal is used.

See Also *ajsgwat, ajsgnl, ajsgrd, ajwapr*

Purpose **Wait for Task Signal(s)**

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup

```
unsigned int tsignals;
long      timeout;
int       match;
int       status;
unsigned int sigrecv;
 .
 .
status = ajsgwat(tsignals,  match,  timeout, &sigrecv);
AX               BX[14..0]  BX[15]  DX:CX    BX=
```

Where *tsignals* is a 15-bit mask identifying the task signals of interest. Only the least significant 15 bits of *tsignals* (bits 0 to 14) are used. Set a bit to wait for the corresponding task signal. All other bits must be 0.

match defines the task signal match requirements:

0 Wait for any of the selected task signals to occur.

<> 0 Wait for all of the selected task signals to occur.

timeout is the maximum interval measured in AMX system ticks which the caller is prepared to wait for the task signal match to occur. *Timeout* must be positive. If *timeout* = 0, the caller will wait forever for the specified task signal match. (See Note).

&sigrecv is a pointer to storage for a 15-bit mask identifying the task signals satisfied by this call. If *&sigrecv* = *NULL (0L)*, no task signal information is returned.

Results Interrupts are disabled and then enabled upon return.

Status is returned.

AEROK	=	Call successful
AERTM0	=	Timed out before required signals occurred
AERTMV	=	Invalid timeout interval (<0)

sigrecv identifies the received task signals which satisfied the match criteria or those which had occurred prior to timeout.

If waiting for any of *n* task signals (*match* = 0),

AEROK:	*sigrecv*	identifies which of the *n* task signals occurred.
AERTMO:	*sigrecv*	= 0.

If waiting for all of *n* task signals (*match* = 0),

AEROK:	*sigrecv*	= *tsignals.*
AERTMO:	*sigrecv*	identifies which of the *n* task signals, if any, occurred before the timeout.

Note Some or all of the task signals specified by *tsignals* may already be pending at the time of the *ajsgwat* call. If these pending task signals satisfy the match criteria, the calling task will resume instantly with *status = AEROK* and *sigrecv* set as described.

Upon return from *ajsgwat*, the task's task signals, if any, indicated by *sigrecv* will have been reset. All other task signals remain unaltered.

If *tsignals* = 0 and *timeout* = 0, the calling task will resume instantly with *status = AEROK* and *sigrecv* = 0.

AMX procedures *ajwait* and *ajwatm* are a simplified form of procedure *ajsgwat* in which one predefined private AMX task signal is used.

Delay If *tsignals* = 0 and a timeout interval is specified, the task will unconditionally delay until the interval has expired. No task signal via *ajsgnl* or *ajwake* call can end the delay prematurely.

See Also *ajsgnl, ajsgrd, ajsgres, ajwait, ajwatm*

Purpose **Generate Software Interrupt**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
#include "amx831sd.h"
 .
 .
struct amxregs regarray;
int       inttype;
unsigned int regeax:
 .
 .
regeax = ajsint(inttype, &regarray);
```

or

```
regeax = ajsintq(inttype, &regarray);
```

Where *inttype* is the interrupt type (number 0 to 255).

regarray is a structure containing the processor register values which are required for the particular software interrupt being generated. *Ajsint* requires that all of the segment registers in *regarray* be initialized with valid selectors. *Ajsintq* unconditionally copies the contents of the current hardware segment registers into the segment registers in *regarray*. General register values which are not required by the interrupt procedure do not have to be provided.

If registers *DS* and/or *ES* are not required by the interrupt procedure, set their values to 0 in *regarray*.

The structure *amxregs* is defined in header file *AMX831SD.H* (see Appendix D).

regeax is the value which was in register *AX* when the interrupt procedure ended.

Results Interrupts will be in the state in which they are left by the interrupt procedure when it exits.

All registers in *regarray* reflect the values in the processor registers at the time the interrupt procedure ended.

The real processor registers are only affected at the instant the software interrupt call is made.

Restrictions The flags register in *regarray* can be set prior to calling *ajsint*. Only the least significant 8 bits of the real flags register are altered to match the value specified in *regarray*.

Upon return, all 16 bits of the flags register in *regarray* are valid.

See Also *ajproc*

Purpose **Create a Semaphore**

Used by ■ Task □ ISP □ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
AMXID     semid;
int       value;
char      tag[4];
int       status;
 .
 .
status = ajsmcre(&semid, value, tag);
AX               BX=     BX     [DX:CX]
```

Where *&semid* is a pointer to storage for the semaphore id of the semaphore allocated to the caller.

value is the initial value of the semaphore.
(-1 <= *value* <= 32767)

If *value* is *-1*, then a resource semaphore is created with an initial value of 1, making the resource free for use. Otherwise, a counting semaphore is created with the initial count specified by parameter *value*.

tag is a 4-character name tag for the semaphore.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

AEROK	=	Call successful
AERISV	=	Invalid semaphore value
AERNSB	=	No free semaphore

If the call is unsuccessful, *semid* is undefined.

See Also *ajsmdel, ajsmtag*

Purpose **Delete a Semaphore**

Used by ■ Task □ ISP □ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
AMXID     semid;
int       status;
 .
 .
status = ajsmdel(semid);
AX                BX
```

Where *semid* is the semaphore id of a resource or counting semaphore acquired by a call to *ajsmcre*.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

AEROK	=	Call successful
AERNSS	=	Invalid semaphore id
AERSIU	=	Semaphore is in use One or more tasks may be waiting on the semaphore.

Restriction You must be absolutely certain that no other task, ISP or Timer Procedure is in any way using or about to use the semaphore. Failure to observe this restriction may lead to unexpected and unpredictable faults.

See Also *ajsmcre*

Purpose **Unconditionally Free a Resource Semaphore**

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup

```
AMXID     semid;
int       status;
 .
 .
status = ajsmfre(semid);
AX                BX
```

Where *semid* is the semaphore id of a resource semaphore acquired by a call to *ajsmcre*.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

AEROK	=	Call successful
AERNSS	=	Invalid semaphore id
AERRNY	=	Resource semaphore cannot be released since the calling task does not own the resource.

Note The resource's use count is immediately set to zero and the resource freed.

The resource will immediately be given to the task (if any) which is waiting at the head of the resource semaphore wait queue. Task rescheduling occurs immediately if necessary.

Restriction You must not attempt to free a counting semaphore. Use *ajsmsig* for that purpose.

See Also *ajsmrls, ajsmrsv*

Purpose **Get Use of a Semaphore (no wait)**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
AMXID     semid;
int       status;
.
.
status = ajsmget(semid);
AX                BX
```

Where *semid* is the semaphore id of a counting semaphore acquired by a call to *ajsmcre*.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

AEROK	=	Call successful
AERNSS	=	Invalid semaphore id
AERSBY	=	Semaphore already in use Use of the semaphore is not granted to caller.

Note This procedure returns immediately with an error indication if the semaphore is in use. Tasks can wait for the use of the semaphore by calling *ajsmwat.*

Restriction You must not attempt to get the use of a resource semaphore. Use *ajsmrsv* for that purpose.

See Also *ajsmwat, ajsmsig*

Purpose **Release a Resource Semaphore**

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup

```
AMXID     semid;
int       status;
 .
 .
status = ajsmrls(semid);      /* nested release          */
AX               BX
```

Where *semid* is the semaphore id of a resource semaphore acquired by a call to *ajsmcre*.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

AEROK	=	Call successful
AERNSS	=	Invalid semaphore id
AERRNY	=	Resource semaphore cannot be released since the calling task does not own the resource.

Note The resource's use count is decremented by one for each call to *ajsmrls*. The resource is not freed until the use count becomes zero.

Once freed, the resource will immediately be given to the task (if any) which is waiting at the head of the resource semaphore wait queue. Task rescheduling occurs immediately if necessary.

Restriction You must not attempt to release a counting semaphore. Use *ajsmsig* for that purpose.

See Also *ajsmfre, ajsmrsv*

Purpose **Reserve a Resource Semaphore**

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup

```
AMXID     semid;
long      timeout;
unsigned int priority;
int       status;
 .
 .
status = ajsmrsv(semid, timeout, priority);
AX               BX     DX:CX    DI
```

Where *semid* is the semaphore id of a resource semaphore acquired by a call to *ajsmcre*.

timeout is the maximum interval measured in AMX system ticks which the caller is prepared to wait for the resource. *Timeout* must be positive. If *timeout* = 0, the caller will wait forever for the resource.

priority is the priority at which the caller wishes to wait (0 = highest). To wait in FIFO order, have all callers use the same value for *priority*.

Results Interrupts are disabled and then enabled upon return.

Status is returned.

AEROK	=	Call successful
AERNSS	=	Invalid semaphore id
AERTMO	=	Timed out without being granted the resource
AERTMV	=	Invalid timeout interval (<0)

The calling task will be suspended waiting for the resource if the resource is owned by some other task.

When the resource is finally available for use by the caller, the procedure will return with *status* = *AEROK*.

Upon first acquiring ownership of the resource, the resource's use count is set to one.

It is permissible for a task to call *ajsmrsv* even if the calling task already owns the resource. The resource use count is incremented by one for each subsequent call by the resource owner.

Restriction You must not attempt to reserve a counting semaphore. Use *ajsmwat* or *ajsmget* for that purpose.

See Also *ajsmrls, ajsmfre*

Purpose **Signal a Semaphore**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
AMXID     semid;
int       status;
 .
 .
status = ajsmsig(semid);
AX                BX
```

Where *semid* is the semaphore id of a counting semaphore acquired by a call to *ajsmcre*.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

AEROK	=	Call successful
AERNSS	=	Invalid semaphore id
AERNME	=	Cannot signal the semaphore because no AMX message envelopes are available for use

The semaphore will immediately be given to the task (if any) which is waiting at the head of the semaphore wait queue. Task rescheduling occurs immediately if necessary.

Restriction You must not attempt to signal a resource semaphore. Use *ajsmfre* or *ajsmrls* for that purpose.

See Also *ajsmwat*

Purpose **Find a Semaphore**

Used by ■ Task □ ISP □ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
AMXID     semid;
char      tag[4];
int       status;
 .
 .
status = ajsmtag(&semid, tag);
AX                BX=     [DX:CX] see note
```

Where *semid* is a pointer to storage for the semaphore id of the semaphore of interest.

tag is a pointer to a 4-character name identifying the semaphore of interest.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

AEROK	=	Call successful
AERNSS	=	No semaphore with matching tag can be found

If more than one semaphore was created with the same tag, you will get back the semaphore id of one of them, but which one is not certain.

If a semaphore with the given tag cannot be located, the semaphore id in *semid* will be undefined.

Semaphore tags are 4 characters long. All characters must match your search tag exactly.

Note A tag *'ABCD'* is presented in register *DX:CX* with *'A'* in *CL* and *'D'* in *DH*.

See Also *ajsmcre*

Purpose **Wait on a Semaphore**

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup

```
AMXID     semid;
long      timeout;
unsigned int priority;
int       status;
 .
 .
status = ajsmwat(semid, timeout, priority);
AX               BX     DX:CX    DI
```

Where *semid* is the semaphore id of a counting semaphore acquired by a call to *ajsmcre*.

timeout is the maximum interval measured in AMX system ticks which the caller is prepared to wait for the semaphore. *Timeout* must be positive. If *timeout* = 0, the caller will wait forever for the semaphore.

priority is the priority at which the caller wishes to wait (0 = highest). To wait in FIFO order, have all callers use the same value for *priority*.

Results Interrupts are disabled and then enabled upon return.

Status is returned.

AEROK	=	Call successful
AERNSS	=	Invalid semaphore id
AERTMO	=	Timed out without being granted use of the semaphore
AERTMV	=	Invalid timeout interval (<0)

Restriction You must not attempt to wait on a resource semaphore. Use *ajsmrsv* for that purpose.

See Also *ajsmget, ajsmsig*

Purpose **Set Pointer Offset**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
unsigned int ofs;
char FAR *pntr;
 .
 .
ajsofs(&pntr, ofs);
```

Results Interrupts are untouched.

The offset part of the *FAR* pointer variable *pntr* is set to the 16-bit unsigned value of *ofs*.

See Also *ajsseg, ajgofs, ajgseg*

Purpose **Set Pointer Segment**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
unsigned short int seg;
char FAR *pntr;
 .
 .
ajsseg(&pntr, seg);
```

Results Interrupts are untouched.

The segment selector part of the *FAR* pointer variable *pntr* is set to the 16-bit unsigned value of *seg*.

See Also *ajssof, ajgofs, ajgseg*

Purpose **Set Segment Registers**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
#include "amx831sd.h"
 .
 .
struct amxsregs sregarray;
 .
 .
ajssreg(&sregarray);
```

Where *&sregarray* is a pointer to a structure defining the selector values to be set in segment registers *DS* and *ES*. Structure *amxsregs* is defined in header file *AMX831SD.H* (see Appendix D).

Results Interrupts are untouched.

Restriction This procedure does NOT modify segment registers *CS* or *SS* to reflect the values in structure *sregarray*.

See Also *ajgsreg*

Purpose **Suspend a Task**

Used by ■ Task □ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
AMXID     taskid;
int       status;
 .
 .
status = ajsusp(taskid);
AX              DX
```

Where *taskid* is the task id of the task to be suspended.

Results Interrupts are disabled and then restored to their state at the time of the call.

The task will be suspended until some other task, ISP or Timer Procedure issues an *ajresum* call to resume this task.

Status is returned.
AEROK = Call successful
AERNST = Invalid task id

Note A task can suspend itself using *ajsusp*. However, it is preferred that you use *ajwait* for this purpose.

See Chapter 13.5 for a description of the intended purpose and use of this procedure.

Restriction You must not suspend the AMX Kernel Task.

See Also *ajresum*

Purpose **Format Time and Date as an ASCII String**

Used by ■ Task □ ISP □ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
#include "amx831sd.h"
 .
 .
struct amxtds tdbuf;
char      ascii[26];
int       format;
int       n;
 .
 .
n = ajtdf(&tdbuf, format, ascii);
AX        ES:BX   DL      DS:DI
```

Where *tdbuf* is an AMX time/date structure containing the time and date (returned by *ajtdg*) which is to be formatted. The structure *amxtds* is defined in header file *AMX831SD.H* (see Appendix D).

format is the format specification value described in Chapter 5.5. Only the least significant 8 bits of *format* are used.

ascii is a 26-byte character array into which time and/or date will be formatted as a null ('\0') terminated ASCII string.

Results Interrupts are untouched.

n is the number of characters stored in the *ascii* array, not including the terminating null ('\0').

See Also *ajtdg*

Purpose **Get Current Time and Date**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
#include "amx831sd.h"
 .
 .
struct amxtds tdbuf;
 .
 .
ajtdg(&tdbuf);
      ES:BX
```

Where *&tdbuf* is a pointer to the structure which is to receive the current time and date. The structure *amxtds* is defined in header file *AMX831SD.H* (see Appendix D).

Results Interrupts are disabled and then restored to their state at the time of the call.

See Also *ajtds, ajtdf*

Purpose **Set the Time and Date**

Used by ■ Task □ ISP □ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
#include "amx831sd.h"
 .
 .
struct amxtds tdbuf;
 .
 .
Initialize tdbuf with new time and date.
 .
 .
ajtds(&tdbuf);
      ES:BX
```

Where *&tdbuf* is a pointer to the structure which contains the new time and date. The structure *amxtds* is defined in header file *AMX831SD.H* (see Appendix D).

Results Interrupts are disabled and then restored to their state at the time of the call.

The AMX system time and date is set to that specified by the caller in structure *tdbuf*.

The values for time and date supplied by the caller are not checked for validity.

See Also *ajtdg*

Purpose **Read System Tick Counter**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
unsigned long tickcnt;
 .
 .
tickcnt = ajtick();
DX:AX
```

Results Interrupts are untouched.

tickcnt is the current value of the AMX system tick counter.

Note The AMX system tick counter is set to 0 when your AMX system is launched. It is incremented once for every elapsed system tick. It wraps from 2^{32}-1 to 0.

Purpose **Create a New Task**

Used by ■ Task □ ISP □ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
#include "amx831sd.h"
 .
 .
struct amxtdts tdt;
AMXID     taskid;
int       status;
 .
 .
status = ajtkcre(&tdt, &taskid);
AX                ES:BX DX=
```

Where *&tdt* is a pointer to the task's definition. All fields in structure *tdt* must be valid before *ajtkcre* is called.

The structure *amxtdts* is defined in header file *AMX831SD.H* (see Appendix D). The fields in the task definition structure correspond to the parameters you must enter in the Task Definition window to predefine a task using the Configuration Manager (see Chapter 14.7).

The task attributes in structure member *amtdtat* must be set to 0 to create a Large model task or to 1 to create a Medium model task. If you create a Medium model task, the task's stack MUST be located in the *DGROUP* data segment.

&taskid is a pointer to storage for the task id of the created task.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

AEROK	=	Call successful
AERNTC	=	No free Task Control Block available
AERITP	=	Invalid task priority (0 or > 127)
AERMBZ	=	Invalid task mailbox depth (<0 or > 32767)

If the task cannot be created, *taskid* will be undefined.

See Also *ajtkdel*

Purpose **Delete a Task**

This procedure removes a task from your AMX system. Its task id will no longer be valid.

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup

```
AMXID     taskid;
int       priority;
int       status;
 .
 .
status = ajtkdel(taskid, priority);
AX               DX      CX
```

Where *taskid* is the task id of the task to be deleted.

priority is the task execution priority at which the deletion is to occur. The deletion priority must be lower than the priority of any task which can affect the task being deleted. The deletion priority must be higher than the priority of any permanently active compute bound task.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

AEROK	=	Call successful
AERNST	=	Invalid task id
AERANA	=	Task deletion is not allowed The task has not yet given its permission to be deleted by calling *ajtktrm* to specify a Task Termination Procedure.

A task can delete itself.

Restriction You must not delete a task which is waiting, or is about to wait, for a resource or counting semaphore, events in an event group or a message on a message exchange. Failure to observe this restriction may lead to unexpected and unpredictable faults.

See Also *ajtkcre, ajtktrm*

Purpose **Get Task Id of Current Task**

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup

```
AMXID     taskid;
 .
 .
taskid = ajtkid();
AX
```

Where *taskid* is the task id of the currently executing task.

Results Interrupts are untouched.

See Also *ajtktcb, ajtktag*

Purpose **Kill a Task**

This procedure will force a ready, executing or suspended task to end. All messages in the task's mailboxes at the time of the kill request will be flushed. All requests for task execution pending at the time of the kill request will be erased.

Used by ■ Task ■ ISP ■ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup

```
AMXID     taskid;
int       status;
 .
 .
status = ajtkill(taskid);
AX                DX
```

Where *taskid* is the task id of the task to be killed.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

AEROK	=	Call successful
AERNST	=	Invalid task id
AERANA	=	Task termination is not allowed The task has not yet given its permission to be killed by calling *ajtktrm* to specify a Task Termination Procedure.

Any task which is waiting for the task being killed to acknowledge receipt of its message sent via *ajsenw* will be allowed to resume execution.

A task can kill itself.

Restriction You must not kill a task which is waiting, or is about to wait, for a resource or counting semaphore, events in an event group or a message on a message exchange. Failure to observe this restriction may lead to unexpected and unpredictable faults.

See Also *ajtkstp, ajtkdel, ajtktrm*

Purpose **Change Task Priority**

Used by ■ Task □ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
AMXID     taskid;
int       priority;
int       status;
 .
 .
status = ajtkpry(taskid, priority);
AX                DX      CX
```

Where *taskid* is the task id of the task whose priority is to be changed.

priority is the desired priority of the task (1 to 127).

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

AEROK	=	Call successful
AERNST	=	Invalid task id
AERITP	=	Invalid task priority (0 or > 127)

See Also *ajtkcre, ajtkdel*

Purpose **Stop Execution of Task**

This procedure will force a ready, executing or suspended task to end.

Used by ■ Task ■ ISP ■ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup

```
AMXID     taskid;
int       status;
 .
 .
status = ajtkstp(taskid);
AX                DX
```

Where *taskid* is the task id of the task to be stopped.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

AEROK	=	Call successful
AERNST	=	Invalid task id
AERANA	=	Task termination is not allowed The task has not yet given its permission to be stopped by calling *ajtktrm* to specify a Task Termination Procedure.

If a task is waiting for the task being stopped to acknowledge receipt of its message sent via *ajsenw*, that task will be allowed to resume execution.

A task can stop itself.

Restriction You must not stop a task which is waiting, or is about to wait, for a resource or counting semaphore, events in an event group or a message on a message exchange. Failure to observe this restriction may lead to unexpected and unpredictable faults.

See Also *ajtkill, ajtkdel, ajtktrm*

Purpose **Get Status of a Task**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
#include "amx831sd.h"
 .
 .
struct amxsbs tasksts;
AMXID     taskid;
int       status;
 .
 .
status = ajtksts(taskid, &tasksts);
AX               DX      ES:BX
```

Where *taskid* is the task id of the task for which status is required.

&tasksts is a pointer to storage for the task status.

> The structure *amxsbs* is defined in header file *AMX831SD.H* (see Appendix D).

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

AEROK	=	Call successful
AERNST	=	Invalid task id

If the call is unsuccessful, the task status in *tasksts* is undefined.

Purpose **Find a Task**

Used by ■ Task □ ISP □ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
char      tag[4];
AMXID     taskid;
int       status;
 .
 .
status = ajktag(&taskid, tag);
AX               DX=      [CX:DX] see note
```

Where *&taskid* is a pointer to storage for the task id of the task of interest.

tag is a pointer to a 4-character name identifying the task of interest.

Results Interrupts are untouched.

Status is returned.
AEROK = Call successful
AERNST = No task with matching task tag can be found

If more than one task was created with the same task tag, you will get back the task id of one of them, but which one is not certain.

If the task with the given tag cannot be located, the task id in *taskid* will be undefined.

Task tags are 4 characters long. All characters must match your search tag exactly.

Note A tag *'ABCD'* is presented in register *DX:CX* with *'A'* in *CL* and *'D'* in *DH*.

See Also *ajtkid, ajtktcb, ajtkcre*

Purpose **Get Task Control Block Pointer**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
#include "amx831sd.h"
 .
 .
struct amxtcbs *tcbp;
AMXID     taskid;
int       status;
 .
 .
status = ajtktcb(taskid, &tcbp);
AX                DX      ES:BX=
```

Where *taskid* is the task id of the task for which the Task Control Block pointer is required.

&tcbp is a pointer to storage for the pointer to the Task Control Block.

The structure *amxtcbs* is defined in header file *AMX831SD.H* (see Appendix D).

Results Interrupts are untouched.

If the *taskid* is invalid, *NULL* (*0L*) is returned in *tcbp*.

Status is returned.
AEROK = Call successful
AERNST = Invalid task id

Do not use the Task Control Block pointer provided by this procedure if there is any chance that the corresponding task could be deleted while the TCB pointer is in your possession.

See Also *ajtkid, ajtktag, ajtkdel*

Purpose **Enable/Disable External Task Termination**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
AMXID     taskid;
void      ttproc();
int       status;
 .
 .
status = ajktrm(taskid, ttproc);
AX               DX      ES:BX
```

Where *taskid* is the task id of the task whose Termination Procedure is to be set.

ttproc is a pointer to the Task Termination Procedure which is to be executed by the task whenever the task is stopped, killed or deleted (see *ajtkstp, ajtkill, ajtkdel*).

If *ttproc* = *NULL* (*0L*), termination of the task is inhibited until a valid Task Termination Procedure is set with *ajtktrm.*

Results Interrupts are untouched.

Status is returned.

AEROK	=	Call successful
AERNST	=	Invalid task id
AERANA	=	Installation is not allowed The task is in the process of being stopped, killed or deleted.

See Also *ajtkdel, ajtkstp, ajtkill*

Purpose **Convert Milliseconds to System Ticks**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
unsigned long ms;
long      ntick;
 .
 .
ntick = ajtmcnv(ms);
DX:CX           DX:CX
```

Where *ms* is the number of milliseconds.

ntick is the equivalent interval in AMX system ticks.

Results Interrupts are untouched.

ntick is the interval value in system ticks.

If the clock frequency is such that *ms* milliseconds requires more than 2^{31} system ticks, *ntick* will be set to 2^{31}-1.

If the clock frequency is such that *ms* milliseconds is less than one half of a system tick, *ntick* will be set to 0.

Note This procedure is frequently used when starting a timer as in the following examples:

```
ajtmwr(ajtmcnv(ms));
ajwatm(ajtmcnv(ms));
```

Avoid using this procedure in ISPs or Timer Procedures where execution speed is critical. For example, if an ISP must start a timer, use *ajtmcnv* during initialization to derive *ntick*. The ISP can then use *ntick* to start the timer without the execution penalty imposed by *ajtmcnv*.

Purpose **Create an Interval Timer**

Used by ■ Task □ ISP □ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
void      tproc();
AMXID     timerid;
long      tperiod;
struct tuser *tparam;
char      tag[4];
int       status;
 .
 .
status = ajtmcre(&timerid, tperiod, tproc, tparam, tag);
AX               DX=       ES:BX is A(Timer Definition Structure)
```

See structure *AMXTMS* defined in header file *AMX831SD.DEF* (see Appendix D).

Where *&timerid* is a pointer to storage for the timer id of the timer allocated to the caller.

tperiod is the timer interval (in AMX system ticks) to be used if the timer is periodic. *Tperiod* must be positive. *tperiod* = 0 for a one-shot timer.

tproc is a pointer to the Timer Procedure to be executed whenever the timer expires.

tparam is a 4-byte parameter which will be passed to the Timer Procedure whenever it is called by AMX. It is usually a pointer to some application variable or structure such as *tuser* in the above example.

tag is a 4-character name tag for the timer.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

AEROK	=	Call successful
AERNTM	=	No free timer
AERTMV	=	Invalid period (<0)

If the call is unsuccessful, *timerid* is undefined.

Note Creating an interval timer does not automatically start the timer. Use *ajtmwr* to start the timer.

See Also *ajtmdel, ajtmtag*

Purpose **Delete an Interval Timer**

Used by ■ Task □ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
AMXID     timerid;
 .
 .
status = ajtmdel(timerid);
AX                DX
```

Where *timerid* is the timer id identifying the timer to be deleted.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.
AEROK = Call successful
AERNSTM = Invalid timer id

Note A Timer Procedure can delete the interval timer with which it is associated.

An active interval timer can be deleted.

Restriction You must be absolutely certain that no other task, ISP or Timer Procedure is in any way using or about to use the interval timer. Failure to observe this restriction may lead to unexpected and unpredictable faults.

See Also *ajtmcre*

Purpose **Read the Current Value of an Interval Timer**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
AMXID     timerid;
long      tval;
 .
 .
tval = ajtmrd(timerid);
DX:CX          DX
```

Where *timerid* is the timer id identifying the timer whose value is required.

tval is the timer value in system ticks.

Results Interrupts are disabled and then restored to their state at the time of the call.

tval >= 0	Call successful *tval* is the current timer value measured in AMX system ticks. *tval* is the time remaining before the timer expires.
tval < 0	Call not successful (*tval* is an error code)
tval = AERNSTM	Invalid timer id

See Also *ajtmcre, ajtmdel, ajtmwr*

Purpose **Find an Interval Timer**

Used by ■ Task □ ISP □ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
AMXID     timerid;
char      tag[4];
int       status;
 .
 .
status = ajtmtag(&timerid, tag);
AX               DX=       [DX:CX] see note
```

Where *&timerid* is a pointer to storage for the timer id of the interval timer of interest.

tag is a pointer to a 4-character name identifying the timer of interest.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.
AEROK = Call successful
AERNSTM = No timer with matching tag can be found

If more than one timer was created with the same tag, you will get back the timer id of one of them, but which one is not certain.

If an interval timer with the given tag cannot be located, the timer id in *timerid* will be undefined.

Timer tags are 4 characters long. All characters must match your search tag exactly.

Note A tag *'ABCD'* is presented in register *DX:CX* with *'A'* in *CL* and *'D'* in *DH*.

See Also *ajtmcre*

Purpose **Start/Stop an Interval Timer**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
AMXID     timerid;
long      tval;
int       status;
 .
 .
status = ajtmwr(timerid, tval);
AX              DX       BX:CX
```

Where *timerid* is the timer id identifying the timer to be started or stopped.

tval is the timer interval measured in AMX system ticks. *Tval* must be positive. Set *tval* to 0 to stop a timer.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

AEROK	=	Call successful
AERNSTM	=	Invalid timer id
AERTMV	=	Invalid interval (<0)

Note A Timer Procedure associated with a periodic timer can stop the timer by calling *ajtmwr* with *tval* = 0. The periodic timer can be restarted by calling *ajtmwr* with *tval* = *n*. The timer will expire after *n* system ticks and resume periodic operation with its predefined period.

Non periodic timers (one-shots) can be retriggered by their associated Timer Procedure using *ajtmwr* to restart the timer.

See Also *ajtmcre, ajtmrd*

Purpose **Trigger a Task**

To request AMX to start a task without sending a message to the task.

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
AMXID     taskid;
int       status;
 .
 .
status = ajtrig(taskid);
AX              DX
```

Where *taskid* is the task id of the task to be triggered.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.
AEROK = Call successful
AERNST = Invalid task id

An immediate task switch will occur if the task being triggered is of higher priority than the current task.

See Also *ajsend, ajend*

Purpose **Change a Task's Time Slice Interval**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
AMXID     taskid;
unsigned int tslice;
int       status;
 .
 .
status = ajtslv(taskid, tslice);
AX              DX      CX
```

Where *taskid* is the task id of the task whose time slice interval is to be altered.

tslice is the new time slice interval in AMX system ticks.
(0 <= *tslice* < 65536)
Set *tslice* to 0 to stop time slicing the task.

Results Interrupts are untouched.

Status is returned.

AEROK	=	Call successful
AERNST	=	Invalid task id

See Also *ajtson, ajtsof*

Purpose **Disable Time Slicing**
Enable Time Slicing

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
ajtsof();                    /* Disable time slicing     */
```

or

```
ajtson();                    /* Enable time slicing      */
```

Results Interrupts are untouched.

See Also *ajtslv*

Purpose **Fetch Pointer to User Parameter Table**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
#include "amx831sd.h"
 .
 .
struct amxupts FAR *uptp; /* User Parameter Table pointer */
 .
 .
ajupt(&uptp);
```

Where *&uptp* is a *FAR* pointer to storage for a *FAR* pointer to your system's User Parameter Table.

The structure *amxupts* is defined in header file *AMX831SD.H* (see Appendix D).

Results Interrupts are untouched.

Note This function must be called from your *main()* program to fetch the User Parameter Table pointer for use in the call to *ajentr()* to launch AMX.

Purpose **Get AMX Version Number**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

```
unsigned long version;
 .
 .
version = ajver();
DX:AX
```

Where *version* is the AMX version number in the hexadecimal format
0x0086rrmm where
rr is the major revision number (*03*)
mm is the minor revision number (*00*)

Results Interrupts are untouched.

Purpose **Wait Unconditionally for a Wake Request**

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup *ajwait();*

Results Interrupts are enabled.

The task will be suspended until some other task, ISP or Timer Procedure issues an *ajwake* call to wake this task.

If a task has an outstanding wake request pending when it calls *ajwait*, the task will continue execution immediately without waiting.

If there is any possibility that some task, ISP or Timer Procedure has already issued an *ajwake* call to wake this task, you should call *ajwapr* to reset any pending wake request before calling *ajwait*.

Note This procedure provides a simplified form of the task signal wait without timeout (see *ajsgwat*).

See Also *ajwatm, ajwake, ajwapr, ajsgwat*

Purpose **Wake Calling Task (Acknowledge Receipt of Message)**

To allow the current task to wake up the task which sent the message which is being processed by the current task.

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup

```
int       status;
 .
 .
status = ajwakc();
AX
```

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

AEROK	=	Call successful
AERCNW	=	Task which sent the message is not waiting
AERNMT	=	Current task is not processing a received message from another task.

An immediate task switch will occur if the task being wakened is of higher priority than the current task.

Note This procedure is equivalent to a call to *ajwakcs* to return a value of *AEROK* (0) to the calling task.

See Also *ajsenw, ajend, ajwakcs*

Purpose **Wake Calling Task (Acknowledge Receipt of Message with Status)**

To allow the current task to wake up the task which sent the message which is being processed by the current task and return completion status to that task.

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup

```
int       status;
int       result;
 .
 .
status = ajwakcs(result);
AX                CX
```

Where *result* is an integer value (>=0) to be returned to the calling task.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

AEROK	=	Call successful
AERCNW	=	Task which sent the message is not waiting.
AERNMT	=	Current task is not processing a received message from another task.
AERIRS	=	Invalid result status (*result* < 0) The message sender was not wakened.

An immediate task switch will occur if the task being wakened is of higher priority than the current task.

Note Calling this procedure with *result = AEROK* (0) is equivalent to a call to procedure *ajwakc*.

See Also *ajsenw, ajend, ajwakc*

Purpose **Wake a Waiting Task**

To wake up a task known to be waiting because of an *ajwait* or *ajwatm* call.

Used by ■ Task ■ ISP ■ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup

```
AMXID     taskid;
int       status;
 .
 .
status = ajwake(taskid);
AX              DX
```

Where *taskid* is the task id of the task to be wakened.

Results Interrupts are disabled and then restored to their state at the time of the call.

Status is returned.

AEROK	=	Call successful
AERWKP	=	Warning: Task was not waiting Wake request is now pending
AERNST	=	Invalid task id
AERTNW	=	Task was not waiting and a wake request was already pending before this *ajwake* call

An immediate task switch will occur if the task being wakened is of higher priority than the current task.

Note This procedure provides a simplified form of task signalling (see *ajsgnl*).

See Also *ajwait, ajwatm, ajwapr, ajsgnl*

Purpose **Reset a Pending Wake Request**

This call only affects the calling task.

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup *ajwapr();*

Results Interrupts are untouched.

Note This procedure provides a simplified form of task signal reset (see *ajsgres*).

See Also *ajwait, ajwatm, ajwake, ajsgres*

Purpose **Task Delay or Timed Wait for a Wake Request**

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup

```
long      tval;
int       status;
 .
 .
status = ajwatm(tval);
AX               DX:CX
```

Where *tval* is the required wait interval or delay period measured in AMX system ticks. *Tval* must be positive. If *tval* = 0, the caller will not wait.

Results Interrupts are enabled.

Status is returned.

AEROK	=	Call successful Task waited for entire interval
AERWBT	=	Task wakened before interval expired
AERTMV	=	Invalid interval (<0)

The caller is allowed to resume execution and its task timer is stopped as soon as another task, ISP or Timer Procedure issues an *ajwake* call specifying this task.

If a task has an outstanding wake request pending when it calls *ajwatm*, the task will continue execution immediately without waiting.

If there is any possibility that some task, ISP or Timer Procedure has already issued an *ajwake* call to wake this task, you should call *ajwapr* to reset any pending wake request before calling *ajwatm*.

Note This procedure provides a simplified form of the task signal wait with timeout (see *ajsgwat*).

If you want to do a task delay with no *ajwake* interference, use *ajsgwat* with a task signal mask of 0.

See Also *ajwait, ajwake, ajwapr, ajsgwat*

This page left blank intentionally.

A. AMX 86 Reserved Words

ajpppppp	AMX C procedure name *pppppp*
AAPPPPPP	AMX assembly language procedure name *PPPPPP*
amxttttt	AMX C structure name of type *ttttt*
AMXID	AMX object identifier (handle)
cdecl	Procedures use C parameter passing conventions
AMssssss	Reserved symbols defined in AMX header files
AERxxxxx	AMX Error Code *xxxxx*
AERWxxxx	AMX Warning Code *xxxx*
AERFxxxx	AMX Fatal Exit Code *xxxx*
AA831FFF.XXX *AM831FFF.XXX* *AMX831FF.XXX*	AMX 86 filenames
AJ831FFF.XXX	AMX 86 C Interface filenames
AA832FFF.XXX	AMX 86 PC Supervisor filenames

This page left blank intentionally.

B. AMX 86 Error Codes

AMX error codes are signed integers. Codes less than zero are error codes. Codes greater than zero are warning codes. To assist you during testing, the hexadecimal value of the least significant 16-bits of the error code is listed as it might appear in a register or memory dump.

Mnemonic		Value (dec)	Value (hex)	Meaning
AEROK	*	*0*	*0*	Call successful
AERNST		*-1*	*0xFFFF*	No such task (invalid task id)
AERNME		*-2*	*0xFFFE*	No free message envelope
AERNMB		*-3*	*0xFFFD*	No mailbox (message queue) defined
AERMBF		*-4*	*0xFFFC*	Mailbox (message queue) full
AERWBT	*	*-5*	*0xFFFB*	Wakened before timeout
AERTNW	*	*-6*	*0xFFFA*	Task not waiting (after 2nd wake)
AERCNW	*	*-7*	*0xFFF9*	Calling task not waiting
AERNMT		*-8*	*0xFFF8*	Message not from task *AASEND* call
		-9	*0xFFF7*	reserved
		-10	*0xFFF6*	reserved
		-11	*0xFFF5*	reserved
AERRNY		*-12*	*0xFFF4*	Resource not owned by you (caller)
AERNSP		*-13*	*0xFFF3*	No such buffer pool (invalid buffer pool id)
AERBTS		*-14*	*0xFFF2*	Buffer too small
AERBNU		*-15*	*0xFFF1*	Buffer not in use
AERBUV		*-16*	*0xFFF0*	Buffer use count overflow
AERITP		*-17*	*0xFFEF*	Invalid task priority
AERNTC		*-18*	*0xFFEE*	No free Task Control Block
AERNTM		*-19*	*0xFFED*	No free interval timer
AERANA		*-20*	*0xFFEC*	Task abort (stop, kill, delete) not allowed
AERIPR		*-21*	*0xFFEB*	Cannot read/write IVT (no IVT access provided)
AERNSS		*-22*	*0xFFEA*	No such semaphore (invalid semaphore id)
AERSIU	*	*-23*	*0xFFE9*	Semaphore in use
AERISV		*-24*	*0xFFE8*	Invalid semaphore value
		-25	*0xFFE7*	reserved
		-26	*0xFFE6*	reserved
AERTMO	*	*-27*	*0xFFE5*	Timed out
AERNMG	*	*-28*	*0xFFE4*	No message available
AERCWT	*	*-29*	*0xFFE3*	Calling task still waiting

* Errors so marked are **not** trapped to your User Error Procedure.

AMX 86 Error Codes (continued)

Mnemonic		Value (dec)	Value (hex)	Meaning
AERNBF		*-30*	*0xFFE2*	No buffers defined
AERNFP		*-31*	*0xFFE1*	No free buffer pool
AERNEB		*-32*	*0xFFE0*	No free event group
AEREVU		*-33*	*0xFFDF*	Event group in use
		-34	*0xFFDE*	reserved
		-35	*0xFFDD*	reserved
		-36	*0xFFDC*	reserved
AERMNA	*	*-37*	*0xFFDB*	Memory not available
AERMIB		*-38*	*0xFFDA*	Invalid memory block pointer
AERMNU		*-39*	*0xFFD9*	Memory block not in use
AERMOV		*-40*	*0xFFD8*	Memory block use count overflow
AERNSX		*-41*	*0xFFD7*	No such message exchange (invalid message exchange id)
AERNXB		*-42*	*0xFFD6*	No free message exchange
AERXIU		*-43*	*0xFFD5*	Message exchange in use
AERMBZ		*-44*	*0xFFD4*	Invalid mailbox (message queue) size
AERNSB		*-45*	*0xFFD3*	No free semaphore
AERNSG		*-46*	*0xFFD2*	No such event group (invalid event group id)
AERNSTM		*-47*	*0xFFD1*	No such timer (invalid timer id)
AERTMV		*-48*	*0xFFD0*	Invalid timing interval
AERIRS		*-49*	*0xFFCF*	Invalid result status *(AAWAKCS)*
AERMMV		*-50*	*0xFFCE*	Memory fill exceeds segment limit
AERSBY	*	*-51*	*0xFFCD*	Semaphore is busy
AERITT		*-52*	*0xFFCC*	Invalid task trap interrupt type

***** Errors so marked are **not** trapped to your User Error Procedure.

AMX 86 Warning Codes

Mnemonic		Value (dec)	Value (hex)	Meaning
AERWNB	*	*1*	*0x0001*	No buffer available
AERWKP	*	*2*	*0x0002*	Task not waiting; wake is pending
AERCWT	*	*3*	*0x0003*	Calling task is still waiting

***** Warnings so marked are **not** trapped to your User Error Procedure.

AMX 86 Fatal Exit Codes

Mnemonic		Value (dec)	Value (hex)	Meaning
AERFX1	•	*1*	*0x0001*	Not enough memory in AMX Data Segment
AERFX2	•	*2*	*0x0002*	Divide, overflow, bound error in ISP
AERFX3	•	*3*	*0x0003*	Divide, overflow, bound error occurred: in a Restart Procedure or in a Timer Procedure or in a task with no task trap handler
AERFX4	•	*4*	*0x0004*	Attempt to exit from a permanent AMX system
AERFX5	•	*5*	*0x0005*	reserved
AERFX6	•	*6*	*0x0006*	ROMed AMX kernel received a request which it is not configured to handle or 20/24-bit Memory Manager library mismatch
AERFX7	•	*7*	*0x0007*	Invalid AMX Configuration parameters: User message size not multiple of 4 or < 12 bytes Kernel Stack size < minimum of 128 bytes Interrupt Stack size < minimum of 128 bytes Number of tasks exceeds maximum of 100
AERFX8	•	*8*	*0x0008*	Launch cannot create predefined task or timer: Invalid task definition Maximum number of tasks allowed is too small Invalid timer definition Maximum number of timers allowed is too small
AERFX9	•	*9*	*0x0009*	Nested breakpoint encountered
AERFX10	•	*10*	*0x000A*	Cannot breakpoint on ISP
AERFX11	•	*11*	*0x000B*	Breakpoint Manager cannot install vectors in IVT
				Managers fail during launch:
AERFXSM	•	*12*	*0x000C*	Cannot create predefined semaphore
AERFXEM	•	*13*	*0x000D*	Cannot create predefined event group
AERFXXM	•	*14*	*0x000E*	Cannot create predefined message exchange
AERFXBM	•	*15*	*0x000F*	Cannot create predefined buffer pool

- All fatal errors **are** trapped to your Fatal Exit Procedure.

This page left blank intentionally.

C. Configuration Generator Specifications

C.1 Introduction

If you are not doing your software development on a PC or compatible running Microsoft® Windows®, then you will be unable to use the interactive Configuration Manager for Windows to create and edit your AMX System Configuration Module. You may, however, still be able to use the Configuration Generator to assist you in this process by porting it to your development system as described in Appendix C.4.

As described in Chapter 14.2, the Configuration Generator merges the information from your User Parameter File with a template of a standard System Configuration Module to produce your module. Since the User Parameter File is a text file, you are free to use the text editor of your choice to create and/or edit this file.

The System Configuration Template provided with the Builder has been coded using Microsoft's *MASM* Assembly Language. Some of the language syntax used by Microsoft makes this module incompatible with Intel's ASM86 Macro Assembly Language. If you are not using an assembler from one of the toolsets supported by KADAK to assemble your System Configuration Module, you may have to edit the template file to reflect the capabilities of your particular assembler.

For these reasons, source code of the template file and the Configuration Generator program has been provided with AMX to allow the Configuration Generator to be ported to your software development environment.

To assist you in this process, the specifications for the User Parameter File and the System Configuration Template are presented in this appendix.

C.2 User Parameter File Specification

The User Parameter File is a text file structured as illustrated in Figure C.2-1. The file consists of a sequence of keywords of the form *...XXX* which begin in column one. Each keyword is followed by one or more parameters which you must provide.

```
; Constant definitions
...UPT      NME,UMS,CLP,CLF,NTK,NTM
...STK      ESS,ISS
...OPT      TSLICE,NSM,NEV,NMX,NBP,ROMS,CAT,BKPT
;
; Time/Date Manager selection
...TAD      UPSHED
;
; Your Fatal Exit Procedure and User Error Procedure
...FEX      UPFEX,UPERR
;
; Your task definitions
...TDT      TKPROC,TKTAG,ATTR,STACK,PRIORITY,SLICE,MB0,MB1,MB2,MB3,TKID
;
; Your Restart Procedures
...RRX      UPRRP
;
; Your Exit Procedures
...EXX      UPEXP
;
; Your interval timer definitions
...TMR      TMPROC,PERIOD,PARAM,TMTAG,TMID
;
; Your semaphore definitions
...SEM      SMVAL,SMTAG,SMID
;
; Your event group definitions
...EVG      EVAL,EVTAG,EVID
;
; Your message exchange definitions
...MEX      MBO,MB1,MB2,MB3,MXTAG,MXID
;
; Your buffer pool definitions
...PDT      BPN,BPS,BPTAG,BPID
;
; Memory Manager options
...MAL      UPMAP
;
; Configuration extension parameters
...EXT      X1,X2,DBGA,BPED,BPXD,PCSOPT
```

Figure C.2-1 User Parameter File

The example in Figure C.2-1 uses symbolic names for all of the parameters following each of the keywords. The symbols correspond to the screen fields described in Chapter 14. You are referred to that chapter for detailed descriptions of each of the parameters.

The order of keywords in the User Parameter File is not particularly critical. For convenience, the keywords have been ordered to closely follow the order of the corresponding entries in your System Configuration Module.

The file begins with a set of **constant definitions**.

NME	Number of message envelopes
UMS	Size of AMX message
CLP	AMX clock period (in hardware ticks)
CLF	Hardware clock frequency (hz)
NTK	Maximum number of tasks
NTM	Maximum number of timers
ESS	AMX Kernel Stack size
ISS	AMX Interrupt Stack size
TSLICE	Time slice option (0 = No; 1 = Yes)
NSM	Maximum number of semaphores
NEV	Maximum number of event groups
NMX	Maximum number of message exchanges
NBP	Maximum number of buffer pools
ROMS	AMX is installed in separate ROM (0 = No; 1 = Yes)
CAT	AMX configuration attributes
BKPT	AMX Breakpoint Manager used (0 = No; 1 = Yes)

The **Time/Date Manager** is selected as follows. If you have provided a Time/Date Scheduling Procedure, include its name as illustrated. If you do not have such a procedure, omit the name but keep the line with keyword *...TAD*.

If you do not want to use the Time/Date Manager, delete the line with keyword *...TAD*.

If you have a **Fatal Exit Procedure**, define its name in place of *UPFEX*. If you do not have such a procedure, leave the name blank. Leave the comma present.

If you have a **User Error Procedure**, define its name in place of *UPERR*. If you do not have such a procedure, leave the name blank. Leave the comma present.

Each of your predefined **tasks** must be defined using keyword *...TDT*. The order of these definitions will determine their order of creation by AMX. If you do not wish to predefine any tasks, delete the line with keyword *...TDT*.

The parameters in each task definition are as follows.

TKPROC	Task procedure name
TKTAG	Task tag
ATTR	Attributes
STACK	Task stack size (bytes)
PRIORITY	Task priority
SLICE	Time slice interval
MB0	Mailbox 0 depth
MB1	Mailbox 1 depth
MB2	Mailbox 2 depth
MB3	Mailbox 3 depth
TKID	Task id variable name

Note that the task tag must include exactly four displayable characters. Spaces and tabs are not allowed.

Define all of your **Restart Procedures** in the order in which you wish them to be executed. Note that you are only defining your own procedures. Restart Procedures required by any of the AMX Managers will automatically be installed for you.

Define all of your **Exit Procedures** in the order in which you wish them to be executed. If you have no Exit Procedures, omit the line containing keyword *...EXX*.

Each of your predefined **interval timers** must be defined using the keyword *...TMR*. The order of these definitions will determine their order of creation by AMX. If you do not wish to predefine any timers, delete the line with keyword *...TMR*.

The parameters in each timer definition are as follows:

TMPROC	Timer Procedure name
PERIOD	Timer period in system ticks (0 = oneshot)
PARAM	Parameter variable name
TMTAG	Timer tag
TMID	Timer id variable name

Note that the timer tag must include exactly four displayable characters. Spaces and tabs are not allowed.

Each of your predefined **semaphores** must be defined using the keyword *...SEM*. The order of these definitions will determine their order of creation by AMX. If you do not wish to predefine any semaphores, delete the line with keyword *...SEM*.

The parameters in each semaphore definition are as follows:

SMVAL	Initial semaphore value - 1 for resource semaphore >=0 for counting semaphore
SMTAG	Semaphore tag
SMID	Semaphore id variable name

Note that the semaphore tag must include exactly four displayable characters. Spaces and tabs are not allowed.

Each of your predefined **event groups** must be defined using the keyword *...EVG*. The order of these definitions will determine their order of creation by AMX. If you do not wish to predefine any event groups, delete the line with keyword *...EVG*.

The parameters in each event group definition are as follows:

EVAL	Initial value of 16 event flags (signed decimal number)
EVTAG	Event group tag
EVID	Event group id variable name

Note that the event group tag must include exactly four displayable characters. Spaces and tabs are not allowed.

Each of your predefined **message exchanges** must be defined using the keyword *...MEX*. The order of these definitions will determine their order of creation by AMX. If you do not wish to predefine any message exchanges, delete the line with keyword *...MEX*.

The parameters in each message exchange definition are as follows:

MB0	Mailbox 0 depth
MB1	Mailbox 1 depth
MB2	Mailbox 2 depth
MB3	Mailbox 3 depth
MXTAG	Message exchange tag
MXID	Message exchange id variable name

Note that the message exchange tag must include exactly four displayable characters. Spaces and tabs are not allowed.

Each of your predefined **buffer pools** must be defined using the keyword *...PDT*. The order of these definitions will determine their order of creation by AMX. If you do not wish to predefine any buffer pools, delete the line with keyword *...PDT*.

The parameters in each buffer pool definition are as follows:

BPN	Number of buffers in the buffer pool
BPS	Size in bytes of each buffer
BPTAG	Buffer pool tag
BPID	Buffer pool id variable name

Note that the buffer pool tag must include exactly four displayable characters. Spaces and tabs are not allowed.

If you intend to use the **Memory Manager**, replace symbol *UPMAP* with the name of your Memory Assignment Procedure.

If you do not wish to use the Memory Manager, omit the line containing keyword *...MAL*.

The **configuration extensions** specified by directive *...EXT* are defined in the toolset dependent chapters of the AMX 86 Tool Guide. Parameters *BPED* and *BPXD* provide the breakpoint entry and exit delays for the Breakpoint Manager. Parameter *PCSOPT* provides additional attributes for the AMX 86 PC Supervisor.

C.3 System Configuration Template

The System Configuration Template is an assembly language source file which defines a System Configuration Module for any system using AMX and its managers. It is recommended that you list file *AM831CG.CT* and examine it carefully before trying to read this description.

The template file consists of KADAK macro definitions for all of the various components of a System Configuration Module. The macro definitions are followed by code generating instances of these macros. Conditional assembly is used to inhibit elements in the module which are not required if the corresponding AMX option has not been selected.

A KADAK macro consists of a macro definition statement, a body of source language statements and directives and a macro end statement. The macro definition and end statements are a special pair of directives defined by KADAK.

A simple example is shown below.

```
;~...MAC     ...TMR   PROC,PERIOD,PARAM,TAG,TMID
             EXTRN    ~1~:FAR              ;Timer Procedure
             IF       ~P3~
             EXTRN    ~3~:FAR              ;Timer parameter
             ENDIF
;~...ENDM
```

The macro definition statement contains a macro directive (*[~...MAC*), a parameter keyword (*...TMR*) and an optional comment string (*PROC,PERIOD,etc.*). These three fields are separated by one or more space and/or tab characters.

The macro directive must start in the first column. The first character of the directive must be the comment character used by the assembler. For Intel compatible assemblers, that character is ;.

The second character in the directive is the parameter delimiter character. This can be any printable character but is reserved exclusively for identifying parameters within the body of the macro. It must not appear in any other context within the macro body. For instance, we could not have used the character *:* since it appears in the *EXTRN* directives. We have chosen the character ~ because it is not required in any of the macro definitions in the System Configuration Template.

Immediately after the parameter delimiter character comes the macro definition keyword *...MAC*.

The parameter keyword (*...TMR*) identifies a particular keyword in the User Parameter File which contains the parameter list needed for expansion of the macro body.

The comment string is optional and is normally used to provide a descriptive list of the macro parameters. The comment string is not used in the parameter substitution process.

Macro parameters appear in the macro body as parameter identifiers. A parameter identifier is a decimal integer preceded and followed by the delimiter character defined in the macro definition statement (*~1~*, *~2~*, etc.). The number identifies the parameter in the particular parameter list specified by the parameter keyword. The parameter is copied from the User Parameter File into the source language macro body replacing the particular identifier. The parameters are numbered 1 to n from left to right in the parameter list following the keyword in the User Parameter File.

A second form of macro parameter has the letter *P* preceding the number in the parameter identifier, as in the *IF* statement in the example. This form directs the Configuration Generator to replace the parameter identifier with the character 1 if the identified parameter is present in the parameter list. If the parameter is missing from the parameter list, the Configuration Generator replaces the parameter identifier with the character 0.

In the above example the macro requires the first and third parameter values from a line in the User Parameter File containing the keyword *...TMR*. If the third parameter is missing from the keyword parameter list in the User Parameter File, the *IF* statement will be false, inhibiting the assembler directive *EXTRN* which, without valid arguments, would produce an assembler error.

The macro is terminated by the macro end directive containing the macro keyword *...ENDM*.

When the Configuration Generator encounters a macro definition in the System Configuration Template File, it scans the User Parameter File for a matching parameter keyword. The macro body is expanded and copied into the System Configuration Module File once for each matching parameter keyword located in the User Parameter File.

A third type of directive is used to define an incremental integer variable. An incremental variable is defined with an initial value and an increment which is added to the variable each time a macro containing the variable is expanded by the Configuration Generator. Note that this *;~...VAR* directive begins with the assembler comment character and the parameter delimiter character required by all KADAK macro directives. The variable name is a string of printable characters preceded and followed by one or more space and/or tab characters. Although only one incremental variable is allowed, its name, initial value and increment can be redefined and used with other macros.

The example below illustrates the use of an incremental variable.

```
;~...VAR    &TN      1,1          ;start at task 1
;~...MAC    ...TDT   PROC,TAG,ATTR,STACK,PR,SLICE,MB0,MB1,MB2,MB3,TKID
;
           DD       (~4~+3)/4 DUP(?)
@L@K&TN    LABEL    WORD         ;Initial stack pointer
;                                ;for task &TN (tag ~2~)
;~...ENDM
```

The first statement in this example is a directive defining an incremental variable named *&TN*. Both its initial value and increment are 1. Note that this directive begins with the assembler comment character (*;*) and the parameter delimiter character (~ in this example) required by all KADAK macro directives. The variable name is a string of printable characters preceded and followed by one or more space and/or tab characters.

Each time the Configuration Generator expands the macro, the current value of the incremental variable is substituted for the variable name wherever it appears within the macro body. For example, assume that the fifth task definition in the User Parameter File is listed as follows:

```
...TDT      taskproc5,TSK5,0,2048,20,0,0,0,0,0,task5id
```

When the Configuration Generator encounters this task definition, the current value of the incremental variable *&TN* will be 5. Therefore the body of the macro will be expanded as follows:

```
;
            DD      (2048+3)/4 DUP(?)
@L@K5       LABEL   WORD        ;Initial stack pointer
;                               ;for task 5 (tag TSK5)
```

Note that the value of the incremental variable is not updated by its increment until after the macro end directive is encountered.

To gain a better insight to this whole process, run the Configuration Manager on a PC even if you are not doing development on a PC. Use it to create a simple AMX system with two or three predefined tasks. Use the manager to generate the System Configuration Module. Then view the User Parameter File and the System Configuration Module produced by the Configuration Manager. Compare the latter file with the System Configuration Template to see exactly how the parameter identifiers have been replaced by parameters from your User Parameter File.

C.4 Porting the Configuration Generator

The Configuration Manager uses the Configuration Generator to generate your System Configuration Module. If you are not doing your development on a PC or compatible, you may wish to port the Configuration Generator to your development system.

The Configuration Generator is a utility program provided with AMX. The program is written in C with portability in mind. Source file *AM831CG.C* is delivered with AMX.

To port the Configuration Generator to your development system, compile the module *AM831CG.C* using your C compiler. Link the resulting object module with your C Run-Time Library to produce a version of the Configuration Generator which will execute in your environment.

The command line syntax to run the Configuration Generator is as follows.

```
AM831CG upfname ctfname scfname -q -w -bsss -esss
```

upfname	User Parameter File filename
ctfname	System Configuration Template filename
scfname	System Configuration Module filename

All three filenames must be present in the order specified. Each must include the full path to the file if the file is not in the current directory. Each filename must include both name and extension.

Command line switches beginning with - are optional and must follow the filenames.

The *-q* (quiet) switch, if present, will inhibit the display of all informational or error messages on the standard output device (*stdout*). This switch is used by the Configuration Manager to inhibit the Configuration Generator from writing to the display screen when it is invoked by the manager.

The Configuration Generator will ignore a KADAK macro in the template file if the User Parameter File contains no instances of the macro keyword. By default, there is no indication in the output file that the unused macro has been stripped. If the *-w* switch is used, the Configuration Generator will echo the macro definition statement to the output file and follow it with the comment "*Unused in this configuration*".

By default, the Configuration Generator uses the macro identification character (*;* in our example) from the macro directive as a comment character to identify the comments inserted into the output file in response to the *-w* switch.

This default assumption can be overridden using the *-b* and *-e* switches to define alternate comment delimiter strings. The strings *sss* can be up to 16 characters long. Special characters must be inserted using the character pair *^x* for each special character. Special characters are:

^b	blank (space)	(*ASCII 0x20*)
^t	tab	(*ASCII 0x09*)
^r	return	(*ASCII 0x0D*)
^n	new-line	(*ASCII 0x0A*)
*^^*	the character ^	(*ASCII 0x5E*)

Since the System Configuration Module file *scfname* will be an assembly language source file, you should always use the switch *-b;* with the *-w* switch to force the Configuration Generator's comment strings to begin with the assembler comment character (*;*) in case a macro identification character other than *;* is used in macro directives within the template file.

For other versions of AMX, the System Configuration Module file *scfname* is a C source file. For these products, you would use the switches *-b/** and *-e*/* with the *-w* switch to force the Configuration Generator's comment strings to begin and end with the C comment delimiter strings */** and **/*.

This page left blank intentionally.

D. AMX 86 Structure and Constant Definitions

D.1 AMX C Structures and Constants

AMX Launch Parameter (see *AAENTR*)

```
#define   AMLPTMP   1                   /* Temporary launch           */
                                        /* Default is permanent launch*/

#define   AMLPVA    2                   /* Vector table is alterable  */
                                        /* Default is non alterable   */
                                        /* table                      */

#define   AMLPIE    4                   /* Launch with interrupts     */
                                        /* enabled.  Default is       */
                                        /* interrupts disabled        */
```

AMX Task Definition Structure

```
struct amxtdts {
          void      (*amtdtst)();       /* task procedure pointer     */
          char      amtdtag1;           /* 4 char task tag            */
          char      amtdtag2;
          char      amtdtag3;
          char      amtdtag4;
          unsigned int *amtdtsp;        /* task stack pointer         */
          unsigned int amtdtskz;        /* size of task stack (bytes) */
          unsigned short int amtdtat;   /* task attributes            */
          short int amtdtpr;            /* task priority              */
          unsigned short int amtdtslc;  /* task time slice            */
                                        /* (system ticks)             */
          short int amtdtmb0;           /* mailbox level 0 depth      */
          short int amtdtmb1;           /* mailbox level 1 depth      */
          short int amtdtmb2;           /* mailbox level 2 depth      */
          short int amtdtmb3;           /* mailbox level 3 depth      */
          };
```

AMX Task Control Block Structure

```
struct amxtcbs {
          char      amtcbrsv[240];      /* private                    */
          unsigned long amtcbusr[4];    /* user defined               */
          };
```

AMX Task Status Block Structure

```
struct amxsbs {
          AMXID     amsbtid;            /* task id                      */
          char      amsbtag1;           /* task tag                     */
          char      amsbtag2;
          char      amsbtag3;
          char      amsbtag4;
          unsigned long amsbst;         /* task status                  */
          unsigned long amsbsig;        /* pending signals              */
          long      amsbtmr;            /* task timer (time remaining)*/
          unsigned short int amsbslc;   /* time slice                   */
          unsigned short int amsbcc;    /* call count                   */
          unsigned short int amsbatr;   /* task attributes              */
          AMXID     amsbctid;           /* caller's task id             */
                                        /* (-1=ISP; 0=AATRIG)           */
          short int amsbcwat;           /* caller waiting if <> 0       */
          short int amsbpry;            /* task priority                */
          unsigned short int amsbmc0;   /* message count; mailbox 0     */
          unsigned short int amsbmc1;   /* message count; mailbox 1     */
          unsigned short int amsbmc2;   /* message count; mailbox 2     */
          unsigned short int amsbmc3;   /* message count; mailbox 3     */
          };
```

Task Status Masks (fields *amsbst, amsbsig*)

```
#define   AMSBWTMR      0x01            /* timer wait                   */
                                        /* (used with other bits)       */
#define   AMSBWTRG      0x02            /* trigger wait (i.e. idle)     */
#define   AMSBWSEM      0x04            /* semaphore wait               */
#define   AMSBWEVT      0x08            /* event group wait             */
#define   AMSBWXCH      0x10            /* message exchange wait        */
#define   AMSBWSND      0x20            /* message send wait            */
#define   AMSBWSUS      0x40            /* suspended                    */
                                        /* (waiting for resume)         */
#define   AMSBWWAT      0x80            /* waiting for wake             */
#define   AMSBWSIG      0x7FFF0000L     /* 15 task signal waits         */
#define   AMSBWORW      0x80000000L     /* OR-wait on AMSBWSIG bits     */
```

Critical Function Codes (passed to Task Termination Procedure)

```
#define AMCFE       1                 /* ajtkstp() - stop (end) task*/
#define AMCFK       2                 /* ajtkill() - kill task      */
#define AMCFD       3                 /* ajtkdel() - delete task    */
```

AMX Extended Message Parameter Structure

(passed on stack above AMX message when starting a task)

```
struct amxmsgxs {
         AMXID     amxmscid;          /* calling task's id          */
         short int amxmsfn;           /* AMX function code          */
                                      /* (see definitions)          */
         short int amxmsrsv[4];       /* reserved                   */
         };
```

AMX Function Codes (field *amxmsfn*)

```
#define AMXMSFNM    0                /* no message on stack        */
#define AMXMSFMS    1                /* message on stack           */
#define AMXMSFMW    2                /* message on stack;          */
                                     /* sender waiting             */
```

AMX User Parameter Table Structure

```
struct amxupts {
     void      (**ampbrpl)();  /* A(Restart Procedure List)            */
     void      (**ampbepl)();  /* A(Exit Procedure List)               */
     long      ampbcfga;       /* Configuration attributes             */
     short int ampbdgrp;       /* user's DGROUP segment selector       */
     short int ampbnmev;       /* number of message envelopes          */
     short int ampbums;        /* user message size (bytes)            */
     void      *ampbmes;       /* A(Message Envelope Segment)          */
     void      *ampbistk;      /* A(top of AMX private stacks)         */
     unsigned int ampbesz;     /* AMX Kernel Stack size (bytes)        */
     unsigned int ampbisz;     /* AMX Interrupt Stack size (bytes)     */
     unsigned int ampblen;     /* size of AMX storage (bytes)          */
     short int ampbclkp;       /* AMX clock period (in hardware ticks)*/
     short int ampbclkf;       /* hardware clock frequency (hz)        */
     void      (*ampbtds)();   /* A(Time/Date Scheduling Procedure)    */
     void      (*ampbfex)();   /* A(Fatal Exit Procedure)              */
     void      (*ampberr)();   /* A(User Error Procedure)              */

     short int ampbtkmx;       /* maximum number of tasks              */
     short int ampbtkpr;       /* number of predefined tasks           */
     void      *ampbtkdf;      /* A(Task Definition Table)             */

     short int ampbtmmx;       /* maximum number of timers             */
     short int ampbtmpr;       /* number of predefined timers          */
     void      *ampbtmdf;      /* A(Timer Definition Table)            */

     short int ampbsmmx;       /* maximum number of semaphores         */
     short int ampbsmpr;       /* number of predefined semaphores      */
     void      *ampbsmdf;      /* A(Semaphore Definition Table)        */

     short int ampbevmx;       /* maximum number of event groups       */
     short int ampbevpr;       /* number of predefined event groups    */
     void      *ampbevdf;      /* A(Event Group Definition Table)      */

     short int ampbmxmx;       /* maximum number of message exchanges  */
     short int ampbmxpr;       /* number of predefined message exchanges*/
     void      *ampbmxdf;      /* A(Message Exchange Definition Table)*/

     short int ampbbpmx;       /* maximum number of buffer pools       */
     short int ampbbppr;       /* number of predefined buffer pools    */
     void      *ampbbpdf;      /* A(Buffer Pool Definition Table)      */

     int       (*ampbmap)();   /* A(Memory Assignment Procedure)       */
     char      ampbdmn[16];    /* reserved for InSight                 */
     void      *ampbpcdt;      /* A(PC Supervisor Definition Table)    */
     long      ampbdba;        /* Debugger attributes                  */
     short int ampbped;        /* Breakpoint entry delay               */
     short int ampbpxd;        /* Breakpoint exit delay                */
     };
```

Configuration Attributes (field *AMPBCFGA*)

```
#define AMCAMTK 1              /* Some Medium tasks          */
#define AMCAMUP 2              /* Some Medium user procedures*/
#define AMCAS24 4              /* Use 24-bit address space   */
```

AMX Time/Date Structure

```
struct amxtds {
        unsigned char amtdsec;        /* seconds  (0-59)            */
        unsigned char amtdmin;        /* minutes  (0-59)            */
        unsigned char amtdhr;         /* hours    (0-23)            */
        unsigned char amtddy;         /* day      (1-31)            */
        unsigned char amtdmn;         /* month    (1-12)            */
        unsigned char amtdyr;         /* year     (0-99)            */
        unsigned char amtdow;         /* day of week (Mon=1 to Sun=7)*/
        unsigned char amtdcen;        /* 0 if time/date is incorrect*/
                                      /* century if time/date       */
                                      /* is correct                 */
        };
```

AMX Timer Definition Structure

```
struct amxtms {
        void     (*amtmproc)();       /* A(Timer Procedure)         */
        long     amtmper;             /* timer period (system ticks)*/
        void     *amtmparm            /* user defined timeout       */
                                      /* parameter                  */
        char     amtmtag1;            /* timer tag                  */
        char     amtmtag2;
        char     amtmtag3;
        char     amtmtag4;
        };
```

AMX Buffer Pool Definition Structure

```
struct amxbps {
        char      *ambpbp;            /* Memory pointer             */
        unsigned int ambpnb;          /* Number of buffers          */
        unsigned int ambpbs;          /* Size of buffers            */
        char      ambptag1;           /* Buffer pool tag            */
        char      ambptag2;
        char      ambptag3;
        char      ambptag4;
        };
```

AMX List Header Structure (doubly linked lists)

```
struct amxlhs {
         struct amxlhs *amlhhead;      /* list head                  */
         struct amxlhs *amlhtail;      /* list tail                  */
         unsigned int amlhoffs;        /* byte offset to object node */
         };
```

AMX List Node Structure

```
struct amxlns {
         struct amxlns *amlnnext;      /* next node in list          */
         struct amxlns *amlnprev;      /* previous node in list      */
         };
```

AMX Key Node Structure

```
struct amxlks {
         struct amxlks *amlknext;      /* next node in list          */
         struct amxlks *amlkprev;      /* previous node in list      */
         unsigned int amlkkey;         /* node key                   */
         };
```

AMX Register Array Structure

```
struct amxregs {
          unsigned short int amxrf;     /* Flags (LS byte only)       */
          unsigned short int amxrax;    /* Register AX                */
          unsigned short int amxrbx;    /* Register BX                */
          unsigned short int amxrcx;    /* Register CX                */
          unsigned short int amxrdx;    /* Register DX                */
          unsigned short int amxrsi;    /* Register SI                */
          unsigned short int amxrdi;    /* Register DI                */
          unsigned short int amxrbp;    /* Register BP                */
          unsigned short int amxrds;    /* Register DS                */
          unsigned short int amxres;    /* Register ES                */
          };
```

AMX Segment Register Array Structure

```
struct amxsregs {
          unsigned short int amxsrds;   /* Register DS                */
          unsigned short int amxsres;   /* Register ES                */
          unsigned short int amxsrss;   /* Register SS                */
          unsigned short int amxsrcs;   /* Register CS                */
          };
```

AMX Interrupt Descriptor Structure

```
struct amxidts {
          void      (*amxiptr)();       /* Interrupt procedure pointer*/
          unsigned short int amxigate;  /* IDT gate type              */
          short int amxirsv             /* Intel reserved             */
          };
```

Interrupt gate types (field *amxigate*)

See Intel 80286 Programmer's Reference Manual

```
#define AMIDTINT    0x8600              /* 80286 interrupt gate       */
#define AMIDTTRP    0x8700              /* 80286 trap gate            */
```

AMX C Root ISP Structure

```
struct amxisps {
          long amxispfn[8];             /* Root ISP                   */
          };
```

AMX C Jump Buffer Structure

```
struct ajxjbuf {
         unsigned short int xjbretadr; /* ofs(return address)        */
         unsigned short int xjbcs;     /* Register CS                */
         unsigned short int xjbsp;     /* Stack pointer offset       */
         unsigned short int xjbss;     /* Stack pointer segment      */
         unsigned short int xjbbx;     /* Register BX                */
         unsigned short int xjbcx;     /* Register CX                */
         unsigned short int xjbsi;     /* Register SI                */
         unsigned short int xjbes;     /* Register ES                */
         unsigned short int xjbdi;     /* Register DI                */
         unsigned short int xjbds;     /* Register DS                */
         unsigned short int xjbbp;     /* Register BP                */
         unsigned short int xjbrsrv;   /* Reserved                   */
         };
```

D.2 AMX Assembler Structures and Constants

AMX Launch Parameter (see *AAENTR*)

```
AMLPTMP   EQU       1           ;Temporary launch
;                               ;Default is permanent launch
;
AMLPVA    EQU       2           ;Vector table is alterable
;                               ;Default is non alterable table
;
AMLPIE    EQU       4           ;Launch with interrupts enabled
                                ;Default is disabled interrupts
```

AMX Task Definition Structure

```
AMXTDTS   STRUC
;
AMTDTST   DD        ?           ;task procedure pointer
AMTDTAG1  DB        ?           ;4 char task tag
AMTDTAG2  DB        ?
AMTDTAG3  DB        ?
AMTDTAG4  DB        ?
AMTDTSP   DD        ?           ;task stack pointer
AMTDTSKZ  DW        ?           ;size of task stack (bytes)
AMTDTAT   DW        ?           ;task attributes
AMTDTPR   DW        ?           ;task priority
AMTDTSLC  DW        ?           ;task time slice (system ticks)
AMTDTMB0  DW        ?           ;mailbox level 0 depth
AMTDTMB1  DW        ?           ;mailbox level 1 depth
AMTDTMB2  DW        ?           ;mailbox level 2 depth
AMTDTMB3  DW        ?           ;mailbox level 3 depth
;
AMXTDTS   ENDS
```

AMX Task Control Block Structure

```
AMXTCBS   STRUC
;
AMTCBRSV  DD        60 DUP (?)          ;private
AMTCBUSR  DD        4 DUP (?)           ;user defined
;
AMXTCBS   ENDS
```

AMX Task Status Block Structure

```
AMXSBS    STRUC
;
AMSBTID   DW        ?                   ;task id
AMSBTAG1  DB        ?                   ;task tag
AMSBTAG2  DB        ?
AMSBTAG3  DB        ?
AMSBTAG4  DB        ?
AMSBST    DD        ?                   ;task status
AMSBSIG   DD        ?                   ;pending signals
AMSBTMR   DD        ?                   ;task timer (time remaining)
AMSBSLC   DW        ?                   ;time slice
AMSBCC    DW        ?                   ;call count
AMSBATR   DW        ?                   ;task attributes
AMSBIDC   DW        ?                   ;caller's task id
                                        ;(-1=ISP; 0=AATRIG)
AMSBCWAT  DW        ?                   ;caller waiting if <> 0
AMSBPRY   DW        ?                   ;task priority
AMSBMC0   DW        ?                   ;message count; mailbox 0
AMSBMC1   DW        ?                   ;message count; mailbox 1
AMSBMC2   DW        ?                   ;message count; mailbox 2
AMSBMC3   DW        ?                   ;message count; mailbox 3
;
AMXSBS    ENDS
```

Task Status Masks (fields *AMSBST, AMSBSIG*)

```
; low word:
AMSBWTMR  EQU       01H                 ;timer wait
                                        ;(used with other bits)
AMSBWTRG  EQU       02H                 ;trigger wait (i.e. idle)
AMSBWSEM  EQU       04H                 ;semaphore wait
AMSBWEVT  EQU       08H                 ;event group wait
AMSBWXCH  EQU       10H                 ;message exchange wait
AMSBWSND  EQU       20H                 ;message send wait
AMSBWSUS  EQU       40H                 ;suspended (waiting for resume)
AMSBWWAT  EQU       80H                 ;waiting for wake
; high word:
AMSBWSIG  EQU       7FFFH               ;15 task signal waits
AMSBWORW  EQU       8000H               ;OR-wait on AMSBWSIG bits
```

Critical Function Codes (passed to Task Termination Procedure)

```
AMCFE     EQU       1                   ;AATKSTP - stop (end) task
AMCFK     EQU       2                   ;AATKILL - kill task
AMCFD     EQU       3                   ;AATKDEL - delete task
```

AMX Extended Message Parameter Structure

(passed on stack above AMX message when starting a task)

```
AMXMSGXS  STRUC
;
AMXMSCID  DW        ?                   ;calling task's id
AMXMSFN   DW        ?                   ;AMX function code
AMXMSRSV  DW        4 DUP(?)            ;reserved
;
AMXMSGXS  ENDS
```

AMX Function Codes (field *AMXMSFN*)

```
AMXMSFNM  EQU       0                   ;no message on stack
AMXMSFMS  EQU       1                   ;message on stack
AMXMSFMW  EQU       2                   ;message on stack
                                        ;sender waiting
```

AMX User Parameter Table Structure

```
AMXUPTS   STRUC
;
AMPBRPL   DD        ?           ;A(Restart Procedure List)
AMPBEPL   DD        ?           ;A(Exit Procedure List)
AMPBCFGA  DD        ?           ;Configuration attributes
AMPBDGRP  DW        ?           ;user's DGROUP segment selector
AMPBNMEV  DW        ?           ;number of message envelopes
AMPBUMS   DW        ?           ;user message size (bytes)
AMPBMES   DD        ?           ;A(Message Envelope Segment)
AMPBISTK  DD        ?           ;A(top of AMX private stacks)
AMPBESZ   DW        ?           ;AMX Kernel Stack size (bytes)
AMPBISZ   DW        ?           ;AMX Interrupt Stack size(bytes)
AMPBLEN   DW        ?           ;size of AMX storage (bytes)
AMPBCLKP  DW        ?           ;AMX clock period (in hardware ticks)
AMPBCLKF  DW        ?           ;hardware clock frequency (hz)
AMPBTDS   DD        ?           ;A(Time/Date Scheduling Procedure)
AMPBFEX   DD        ?           ;A(Fatal Exit Procedure)
AMPBERR   DD        ?           ;A(User Error Procedure)
;
AMPBTKMX  DW        ?           ;maximum number of tasks
AMPBTKPR  DW        ?           ;number of predefined tasks
AMPBTKDF  DD        ?           ;A(Task Definition Table)
;
AMPBTMMX  DW        ?           ;maximum number of timers
AMPBTMPR  DW        ?           ;number of predefined timers
AMPBTMDF  DD        ?           ;A(Timer Definition Table)
AMPBSMMX  DW        ?           ;maximum number of semaphores
AMPBSMPR  DW        ?           ;number of predefined semaphores
AMPBSMDF  DD        ?           ;A(Semaphore Definition Table)
;
AMPBEVMX  DW        ?           ;maximum number of event groups
AMPBEVPR  DW        ?           ;number of predefined event groups
AMPBEVDF  DD        ?           ;A(Event Group Definition Table)
;
AMPBMXMX  DW        ?           ;maximum number of message exchanges
AMPBMXPR  DW        ?           ;number of predefined message exchanges
AMPBMXDF  DD        ?           ;A(Message Exchange Definition Table)
;
AMPBBPMX  DW        ?           ;maximum number of buffer pools
AMPBBPPR  DW        ?           ;number of predefined buffer pools
AMPBBPDF  DD        ?           ;A(Buffer Pool Definition Table)
;
AMPBMAP   DD        ?           ;A(Memory Assignment Procedure)
;
AMPBDMN   DB        16 DUP(?)   ;reserved for InSight
AMPBPCS   DD        ?           ;A(PC Supervisor Definition Table)
AMPBDBA   DD        ?           ;Debugger attributes
AMPBPED   DW        ?           ;Breakpoint entry delay
AMPBPXD   DW        ?           ;Breakpoint exit delay
;
AMXUPTS   ENDS
```

Configuration Attributes (field *AMBPCFGA*)

```
AMCAMTK   EQU       1                   ;Some Medium tasks
AMCAMUP   EQU       2                   ;Some Medium user procedures
AMCAS24   EQU       4                   ;Use 24-bit address space
```

AMX Time/Date Structure

```
AMXTDS    STRUC
;
AMTDSEC   DB        ?                   ;seconds
AMTDMIN   DB        ?                   ;minutes
AMTDHR    DB        ?                   ;hours
AMTDDY    DB        ?                   ;day
AMTDMN    DB        ?                   ;month
AMTDYR    DB        ?                   ;year
AMTDOW    DB        ?                   ;day of week
AMTDCEN   DB        ?                   ;0 if time/date is incorrect
;                                       ;century if time/date is correct
;
AMXTDS    ENDS
```

AMX Timer Definition Structure

```
AMXTMS    STRUC
;
AMTMPROC  DD        ?               ;A(Timer Procedure)
AMTMPER   DD        ?               ;timer period (system ticks)
AMTMPARM  DD        ?               ;user defined timeout parameter
AMTMTAG1  DB        ?               ;timer tag
AMTMTAG2  DB        ?
AMTMTAG3  DB        ?
AMTMTAG4  DB        ?
;
AMXTMS    ENDS
```

AMX Buffer Pool Definition Structure

```
AMXBPS    STRUC
;
AMBPBP    DD        ?                   ;Memory pointer
AMBPNB    DW        ?                   ;Number of buffers
AMBPBS    DW        ?                   ;Size of buffers
AMBPTAG1  DB        ?                   ;Buffer pool tag
AMBPTAG2  DB        ?
AMBPTAG3  DB        ?
AMBPTAG4  DB        ?
;
AMXBPS    ENDS
```

AMX List Header Structure (doubly linked lists)

```
AMXLHS    STRUC
;
AMLHHEAD  DD        ?                   ;Head of list
AMLHTAIL  DD        ?                   ;Tail of list
AMLHOFFS  DW        ?                   ;Byte offset to object node
;
AMXLHS    ENDS
```

AMX List Node Structure

```
AMXLNS    STRUC
;
AMLNNEXT  DD        ?                   ;Next node in list
AMLNPREV  DD        ?                   ;Previous node in list
;
AMXLNS    ENDS
```

AMX Key Node Structure

```
AMXLKS    STRUC
;
AMLKNEXT  DD        ?                   ;Next node in list
AMLKPREV  DD        ?                   ;Previous node in list
AMLKKEY   DW        ?                   ;Node key
;
AMXLKS    ENDS
```

AMX Register Array Structure

```
AMXREGS   STRUC
;
AMXRF     DW        ?                   ;Flags (LS byte only)
AMXRAX    DW        ?                   ;Register AX
AMXRBX    DW        ?                   ;Register BX
AMXRCX    DW        ?                   ;Register CX
AMXRDX    DW        ?                   ;Register DX
AMXRSI    DW        ?                   ;Register SI
AMXRDI    DW        ?                   ;Register DI
AMXRBP    DW        ?                   ;Register BP
AMXRDS    DW        ?                   ;Register DS
AMXRES    DW        ?                   ;Register ES
;
AMXREGS   ENDS
```

AMX Segment Register Array Structure

```
AMXSREGS  STRUC
;
AMXSRDS   DW        ?                   ;Register DS
AMXSRES   DW        ?                   ;Register ES
AMXSRSS   DW        ?                   ;Register SS
AMXSRCS   DW        ?                   ;Register CS
;
AMXSREGS  ENDS
```

AMX Interrupt Descriptor Structure

```
AMXIDTS   STRUC
;
AMXIPTR   DW        ?                   ;FAR A(ISP) - offset
          DW        ?                   ;           - selector
AMXIGATE  DW        ?                   ;IDT gate type
          DW        ?                   ;Intel reserved
;
AMXIDTS   ENDS
```

Interrupt gate types (field *AMXIGATE*)

See Intel 80286 Programmer's Reference Manual

```
AMIDTINT  EQU       8600H               ;80286 interrupt gate
AMIDTTRP  EQU       8700H               ;80286 trap gate
```

This page left blank intentionally.

E. AMX 86 Assembler Interface

This appendix summarizes the assembly language calling sequences for all AMX procedures. The procedures are organized in functional groups. Within each group the procedures are listed alphabetically.

All procedures are called as follows:

```
EXTRN          AAxxxx:FAR      ;external FAR procedure
.
.
Parameters in registers specified by column IN
CALL           AAxxxx
Results in registers specified by column OUT
AX = AEROK     flags = Z       no error
AX = AERxx     flags = NZ, S   error
AX = AERWxx    flags = NZ, NS  warning
```

For some functions, *AX* returns a parameter and no error status is provided. In such cases, the zero and sign flags are unknown. *AX* is always altered unless otherwise specified.

All registers except *AX* and those used to return results are preserved. Exceptions are noted.

The direction flag is always preserved. Other flags which are altered by arithmetic and logical operations are unknown upon return.

Interrupts are affected as follows:

E	D	R	
□	□	□	Untouched
□	■	□	Disabled upon return
■	□	□	Enabled upon return
■	■	□	Disabled and then enabled upon return
□	■	■	Disabled and then restored to the state at the time of the call
■	■	■	Disabled and enabled and then restored to the state at the time of the call

The following notes are referenced in the procedure descriptions in this appendix.

Note:

1. All registers are preserved, including `AX`.

2. Registers `CX`, `DX`, `SI` are unaltered.
   All other registers are undefined.

3. All registers are restored to the state they were in when `AAINT` was called.

4. Registers `AX`, `BX`, `CX` and `DX` are altered.
   All other registers are preserved.

5. A tag `'ABCD'` is presented in `DX:CX` with `'A'` in `CL` and `'D'` in `DH`.

6. `AX` is `0` and `flags = Z` if `ES:BX` is `0:0`.
   Otherwise `AX` is non-zero and `flags = NZ`.

PROCEDURE C	PROCEDURE ASM	PURPOSE	PARAMETERS	IN	OUT	AX = ERRORS	E	D	R
System Control									
ajentr	AAENTR	Enter AMX multitasking world	Launch Parameters Exit allowed? (0=No, 1=Yes) IVT alterable (0=No, 1=Yes) Interrupts enabled (0=No, 1=Yes) A(AMX User Parameter Table) errcode exitinfo	BX bit 0 bit 1 bit 2 DS:SI	no AX DX:BX		■	■	■
ajexit	AAEXIT	Leave AMX multitasking world (no return)	errcode exitinfo	CX DX:BX		no	■	■	□
ajfatl	AAFATL	Fatal exit (no return)	Fatal Exit Code	CX		no	□	■	□
ajhook	AAHOOK	Install hooks to AMX Scheduler	A(Block of procedure pointers) (0:0 to remove hooks)	ES:BX		no	□	□	□
ajprvl	AAPRVL	Lower privilege level				no	□	■	■
ajprvr	AAPRVR	Raise privilege level				no	□	■	■
ajver	AAVER	Get AMX version number	Major revision rr Minor revision mm AMX version = 0086H	AH AL DX	no		□	□	□

PROCEDURE C	PROCEDURE ASM	PURPOSE	PARAMETERS	IN	OUT	AX = ERRORS	E	D	R
Task Control									
ajend	AAEND	End task execution (no return)				no	■	□	□
ajgmsg	AAGMSG	Get message from task mailbox	A(Storage for message) Mailbox priority (0 to 3) (4=highest priority message)	ES:BX CX		AERCWT AERNMG	□	■	■
ajresum	AARESUM	Resume a task suspended by ajsusp	Task id	DX		AERNST	□	■	■
ajsend ajsendp ajsenw ajsenwp	AASEND	Start a task by sending it a message at one of 4 priorities	Task id A(message) Proceed = 0; Wait = 80H Priority (0 to 3)	DX ES:BX CH CL		AERNST AERNME AERNMB AERMBF	□	■	■
ajsgnl	AASGNL	Signal a task	Task id Signal mask Pending signals Signal overruns	DX BX	 BX CX	AERNST AERWAT AERWKP AERTNW	□	■	■
ajsgrd	AASGRD	Read pending task signals	signals		AX	no	□	□	□
ajsgres	AASGRES	Reset pending task signals	signal mask	BX		no	□	□	□
ajsgwat	AASGWAT	Wait for any/all of a set of task signals	signal mask Wait all = 1; Wait any = 0 Max. wait time (system ticks) (0:0 = forever) signals received	BX BX[15] DX:CX	 BX	AERTMO AERTMV	■	■	□
ajsusp	AASUSP	Suspend a task	Task id	DX		AERNST	□	■	■
ajtkcre	AATKCRE	Create a new task	A(Task Definition) Assigned task id	ES:BX	 DX	AERITP AERNTC AERMBZ	□	■	■
ajtkdel	AATKDEL	Delete a task	Task id Priority for delete	DX CX		AERNST AERANA	□	■	■
ajtkid	AATKID	Get task identifier of current task	Task id		AX	no	□	□	□

PROCEDURE C	PROCEDURE ASM	PURPOSE	PARAMETERS	IN	OUT	AX = ERRORS	E	D	R
Task Control (continued)									
ajtkill	AATKILL	Kill a task	Task id	DX		AERNST AERANA	□	■	■
ajtkpry	AATKPRY	Change task's execution priority	Task id Task's new priority	DX CX		AERNST AERITP	□	■	■
ajtkstp	AATKSTP	Stop a task	Task id	DX		AERNST AERANA	□	■	■
ajtksts	AATKSTS	Fetch task status	Task id A(storage for result)	DX ES:BX		AERNST	□	■	■
ajtktag	AATKTAG	Find task id of task with a specific task tag	Task tag Task id	DX:CX	DX	AERNST Note 5	□	■	■
ajtktcb	AATKTCB	Fetch A(Task Control Block)	Task id A(TCB for that task)	DX	ES:BX	AERNST	□	□	□
ajtktrm	AATKTRM	Enable/disable abnormal task termination	Task id A(Task Termination Procedure) (0:0 to disable termination)	DX ES:BX		AERNST AERANA	□	■	■
ajtrig	AATRIG	Start (trigger) a task with no message	Task id	DX		AERNST	□	■	■
ajwait	AAWAIT	Wait for a wake request				Note 1	■	□	□
ajwakc	AAWAKC	Wake a task upon receipt of its message				AERCNW AERNMT	□	■	■
ajwakcs	AAWAKCS	Wake a task upon receipt of its message Return status to that task	Returned status	CX		AERCNW AERNMT AERIRS	□	■	■
ajwake	AAWAKE	Wake a task	Task id	DX		AERWKP AERNST AERTNW	□	■	■
ajwapr	AAWAPR	Reset pending wake for current task				no	□	□	□
ajwatm	AAWATM	Wait for a timed interval or until a wake request occurs	Number of system ticks to wait	DX:CX		AERWBT AERTMV	■	□	□

PROCEDURE C	PROCEDURE ASM	PURPOSE	PARAMETERS	IN	OUT	AX = ERRORS	E	D	R
Interrupt Control									
ajint	AAINT	Begin interrupt service	AMX Interrupt Stack DGROUP segment		SS:SP DS,ES	Note 2	□	■	□
ajinx	AAINX	End interrupt service				Note 3	□	■	□
ajispm	AAISPM	Make an ISP root	A(Storage for ISP root) A(Interrupt Procedure)	ES:BX CX:DX		no	□	□	□
ajitrp	AAITRP	Install task trap handler	Interrupt type 0,4 or 5 A(task trap handler)	DX ES:BX		AERITT	□	□	□
ajivtr	AAIVTR	Read entry in IVT	Interrupt type (0-255) A(Storage for old vector)	DX ES:BX		AERIPR	□	■	■
ajivtw	AAIVTW	Write entry in IVT	Interrupt type (0-255) A(New ISP)	DX ES:BX		AERIPR	□	■	■
ajivtx	AAIVTX	Exchange entry in IVT	Interrupt type (0-255) A(New ISP) A(Storage for old ISP)	DX ES:BX DS:DI		AERIPR	□	■	■

PROCEDURE C	PROCEDURE ASM	PURPOSE	PARAMETERS	IN	OUT	AX = ERRORS	E	D	R
Timing Control									
ajclk	AACLK	AMX Clock Handler				Note 1	■	■	■
ajtick	AATICK	Read elapsed system ticks			DX:AX	no	□	■	■
ajtmcnv	AATMCNV	Convert milliseconds to system ticks	Interval (ms) Interval (system ticks)	DX:CX	 DX:CX	no	□	□	□
ajtmcre	AATMCRE	Create an interval timer	A(Timer Definition) Timer id	ES:BX	 DX	AERNTM AERTMV Note 5	□	■	■
ajtmdel	AATMDEL	Delete an interval timer	Timer id	DX		AERNSTM	□	■	■
ajtmrd	AATMRD	Read an interval timer	Timer id Timer value (system ticks)	DX	 DX:CX	AERNSTM	□	■	■
ajtmtag	AATMTAG	Find id of timer with specific tag	Tag Timer id	DX:CX	 DX	AERNSTM Note 5	□	■	■
ajtmwr	AATMWR	Start/stop an interval timer	Timer id Timer value (system ticks) (0:0 = stop)	DX BX:CX		AERNSTM AERTMV	□	■	■
ajtslv	AATSLV	Change a task's time slice interval	Task id Time slice value (system ticks)	DX CX		AERNST	□	□	□
ajtsof	AATSOF	Disable time slicing				no	□	□	□
ajtson	AATSON	Enable time slicing				no	□	□	□
Time/Date Manager									
ajtdf	AATDF	Format calendar time/date as an ASCII string	A(time/date structure) A(text buffer) Format specification byte Count of chars formatted	ES:BX DS:DI DL	 AX	no	□	□	□
ajtdg	AATDG	Get calendar time/date	A(time/date structure)	ES:BX		Note 1	□	■	■
	AATDRR	Time/Date Restart Procedure				Note 4	□	■	■
ajtds	AATDS	Set calendar time/date	A(time/date structure)	ES:BX		Note 1	□	■	■

PROCEDURE C	PROCEDURE ASM	PURPOSE	PARAMETERS	IN	OUT	AX = ERRORS	E	D	R
Semaphore Manager									
ajsmcre	AASMCRE	Create a semaphore	Initial value (0 to 32767) (-1 for resource semaphore) Tag Semaphore id	BX DX:CX	 BX	AERISV AERNSB Note 5	□	■	■
ajsmdel	AASMDEL	Delete a semaphore	Semaphore id	BX		AERNSS AERSIU	□	■	■
ajsmfre	AASMFRE	Free a resource (unconditional)	Resource semaphore id	BX		AERNSS AERRNY	□	■	■
ajsmget	AASMGET	Get use of a semaphore (no wait)	Semaphore id	BX		AERNSS AERSBY	□	■	■
ajsmrls	AASMRLS	Release a resource (nested)	Resource semaphore id	BX		AERNSS AERRNY	□	■	■
	AASMRR	Semaphore Manager Restart Procedure				Note 4	□	■	■
ajsmrsv	AASMRSV	Reserve a resource (optional timeout)	Resource semaphore id Max. wait time (system ticks) (0:0 = forever) Wait priority (0 = highest)	BX DX:CX DI		AERNSS AERTMO AERTMV	■	■	□
ajsmsig	AASMSIG	Signal to a semaphore	Semaphore id	BX		AERNSS AERNME	□	■	■
ajsmtag	AASMTAG	Find id of semaphore with specific tag	Tag Semaphore id	DX:CX	 BX	AERNSS Note 5	□	■	■
ajsmwat	AASMWAT	Wait on a semaphore (optional timeout)	Semaphore id Max. wait time (system ticks) (0:0 = forever) Waiting priority (0 = highest)	BX DX:CX DI		AERNSS AERTMO AERTMV	■	■	□

PROCEDURE C	PROCEDURE ASM	PURPOSE	PARAMETERS	IN	OUT	AX = ERRORS	E	D	R
Event Manager									
ajevcre	AAEVCRE	Create an event group	Initial value for event group Tag Group id	BX DX:CX	 BX	AERNEB Note 5	□	■	■
ajevdel	AAEVDEL	Delete an event group	Group id	BX		AERNSG AEREVU	□	■	■
ajevnt	AAEVNT	Get saved event state	Event value		AX	no	□	□	□
ajevrd	AAEVRD	Read current event state	Group id Event value	BX	 CX	AERNSG	□	□	□
	AAEVRR	Event Manager Restart Procedure				Note 4	□	■	■
ajevsig	AAEVSIG	Signal one or more events in a group	PUSH <signal value> PUSH <event mask> PUSH <group id> CALL AAEVSIG ADD SP,6			AERNSG AERNME	□	■	■
ajevtag	AAEVTAG	Find id of event group with specific tag	Tag Group id	DX:CX	 BX	AERNSG Note 5	□	■	■
ajevwat	AAEVWAT	Wait for all/any of a set of events	DX:CX=<timeout-system ticks> (0:0 = forever) PUSH DX PUSH CX PUSH <0 for any; 1 for all> PUSH <match value> PUSH <event mask> PUSH <group id> CALL AAEVWAT ADD SP,12			AERTMO AERNSG AERTMV	■	■	□

PROCEDURE C	PROCEDURE ASM	PURPOSE	PARAMETERS	IN	OUT	AX = ERRORS	E	D	R
Message Exchange Manager									
ajmxcre	AAMXCRE	Create a message exchange	Tag A(Mailbox size definition) DW size mailbox 0 DW size mailbox 1 DW size mailbox 2 DW size mailbox 3 Exchange id	DX:CX ES:BX	 BX	AERNXB AERMBZ Note 5	□	■	■
ajmxdel	AAMXDEL	Delete an exchange	Exchange id	BX		AERNSX AERXIU	□	■	■
ajmxget	AAMXGET	Get a message from an exchange (no wait)	Exchange id A(Storage for message)	BX ES:SI		AERNSX AERNMG	□	■	■
	AAMXRR	Message Exchange Manager Restart Procedure				Note 4	□	■	■
ajmxsnd ajmxsndp	AAMXSND	Send message to exchange	Exchange id Priority (0 to 3) A(Message)	BX DX ES:SI		AERNSX AERMBF AERNME AERNMB	□	■	■
ajmxtag	AAMXTAG	Find id of exchange with specific tag	Tag Exchange id	DX:CX	 BX	AERNSX Note 5	□	■	■
ajmxwat	AAMXWAT	Wait for message from exchange	Exchange id A(Storage for message) Max. wait time (system ticks) (0:0 = forever) Waiting priority (0 = highest)	BX ES:SI DX:CX DI		AERNSX AERTMO AERTMV	■	■	■

PROCEDURE C	PROCEDURE ASM	PURPOSE	PARAMETERS	IN	OUT	AX = ERRORS	E	D	R
Buffer Manager									
ajbau	AABAU	Add to buffer use count	A(buffer) Increment for buffer use count	ES:BX DX		AERBNU AERBUV AERNSP	□	■	■
ajbcre	AABCRE	Create a buffer pool	A(Buffer Pool Definition) Pool id	ES:BX	DX	AERBTS AERNBF AERNFP	□	■	■
ajbdel	AABDEL	Delete a buffer pool	Pool id	DX		AERNSP	□	■	■
ajbfre	AABFRE	Free a buffer	A(buffer)	ES:BX		AERBNU AERNSP	□	■	■
ajbget	AABGET	Get a buffer from a specific pool	Pool id A(buffer)	DX	ES:BX	AERNSP AERWNB	□	■	■
ajbgsz	AABGSZ	Get size of buffer	A(buffer) Size of buffer	ES:BX	CX	AERBNU AERNSP	□	□	□
ajbia	AABIA	Initialize all buffer pools				AERBTS	□	■	■
ajbip	AABIP	Initialize one buffer pool	Pool id	DX		AERNSP AERBTS	□	■	■
	AABMRR	Buffer Manager Restart Procedure				Note 4	□	■	■
ajbtag	AABTAG	Find id of buffer pool with specific tag	Tag Pool id	DX:CX	DX	AERNSP Note 5	□	■	■

PROCEDURE C	PROCEDURE ASM	PURPOSE	PARAMETERS	IN	OUT	AX = ERRORS	E	D	R
Memory Manager									
ajmau	AAMAU	Add to block use count	A(memory block) Increment for use count	ES:BX DX		AERMIB AERMNU AERMOV	□	■	■
ajmfre	AAMFRE	Free a block of memory	A(memory block)	ES:BX		AERMIB AERMNU	□	■	■
ajmgeh	AAMGEH	Get a block of memory from a private section	Required memory size Handle A(allocated memory block) Size of block or available memory	DX:CX ES:BX	ES:BX DX:CX	AERMNA AERMIB	□	■	■
ajmget	AAMGET	Get a block of memory	Required memory size A(allocated memory block) Size of block or available memory	DX:CX	ES:BX DX:CX	AERMNA	□	■	■
ajmgsz	AAMGSZ	Get size of block of memory	A(memory block) Size of memory block	ES:BX	DX:CX	AERMIB AERMNU	□	■	■
ajmhan	AAMHAN	Create private memory section	A(memory block) Size of memory block Handle	ES:BX DX:CX	ES:BX	AERMNA	□	□	□
	AAMMRR	Memory Manager Restart Procedure				Note 4	□	■	■
ajmset	AAMSET	Fill a block of memory with a pattern	A(memory to be filled) Size of memory region Fill pattern	ES:BX DX:CX SI		AERMMV	□	□	□

PROCEDURE C	PROCEDURE ASM	PURPOSE	PARAMETERS	IN	OUT	AX = ERRORS	E	D	R
Linked List Manager									
ajlcre	AALCRE	Create an empty list	A(List Header) Node offset	DS:SI CX		no	□	□	□
ajlhead	AALHEAD	Find head of list	A(List Header) A(First object) (0:0 if list is empty)	DS:SI	ES:BX	Note 6	□	□	□
ajlinsc	AALINSC	Insert before specific object	A(List Header) A(New object) A(Specific object)	DS:SI ES:BX CX:DI		no	□	■	■
ajlinsh	AALINSH	Insert at head of list	A(List Header) A(Object)	DS:SI ES:BX		no	□	■	■
ajlinsk	AALINSK	Insert into keyed list	A(List Header) A(Object) Key	DS:SI ES:BX CX		no	□	■	■
ajlinst	AALINST	Insert at tail of list	A(List Header) A(Object)	DS:SI ES:BX		no	□	■	■
ajlmerg	AALMERG	Merge two lists	PUSH SEGMENT <Source object> PUSH OFFSET <Source object> PUSH SEGMENT <Destination object> PUSH OFFSET <Destination object> PUSH SEGMENT <Source List Header> PUSH OFFSET <Source List Header> PUSH SEGMENT <Destination List Header> PUSH OFFSET <Destination List Header> CALL AALMERG ADD SP,16			no	□	■	■
ajlnext	AALNEXT	Find next object on list	A(List Header) A(Current object) A(Next object) (0:0 if list is empty)	DS:SI ES:BX	ES:BX	Note 6	□	□	□
ajlordk	AALORDK	Reorder object on keyed list	A(List Header) A(Object) Key	DS:SI ES:BX CX		no	□	■	■
ajlprev	AALPREV	Find previous object on list	A(List Header) A(Current object) A(Previous object) (0:0 if list is empty)	DS:SI ES:BX	ES:BX	Note 6	□	□	□

PROCEDURE C	PROCEDURE ASM	PURPOSE	PARAMETERS	IN	OUT	AX = ERRORS	E	D	R
Linked List Manager (continued)									
ajlrmvc	AALRMVC	Remove specific object from list	A(List Header) A(Specific object)	DS:SI ES:BX		no	□	■	■
ajlrmvh	AALRMVH	Remove object at head of list	A(List Header) A(Object) (0:0 if list is empty)	DS:SI	ES:BX	Note 6	□	■	■
ajlrmvt	AALRMVT	Remove object at tail of list	A(List Header) A(Object) (0:0 if list is empty)	DS:SI	ES:BX	Note 6	□	■	■
ajltail	AALTAIL	Find tail of list	A(List Header) A(Last object) (0:0 if list is empty)	DS:SI	ES:BX	Note 6	□	□	□
Circular List Manager									
ajabl	AAABL	Add to bottom of list	Address of list Entry to be added - byte - word - dword	ES:BX CL CX DX:CX		0 ok 1 full -1 no	□	■	■
ajatl	AAATL	Add to top of list	Address of list Entry to be added - byte - word - dword	ES:BX CL CX DX:CX		0 ok 1 full -1 no	□	■	■
ajrbl	AARBL	Remove from bottom of list	Address of list Entry removed - byte - word - dword	ES:BX	CL CX DX:CX	0 ok 1 empty -1 none	□	■	■
ajrstl	AARSTL	Reset a list	Size of slot (1, 2 or 4) Number of slots Address of list	DX CX ES:BX		Note 1	□	■	■
ajrtl	AARTL	Remove from top of list	Address of list Entry removed - byte - word - dword	ES:BX	CL CX DX:CX	0 ok 1 empty -1 none	□	■	■

Numerals

24-bit memory addressing 1, 125, 171

A

AACLK **73**
AAENTR 22
AAINT 54, 55, 56, 73
AAINX 55, 56, 73

ajabl 135, **205**
ajatl 135, **206**
ajbau 52, 119, 121, **207**
ajbcre 119, 120, 174, 189, **208**
ajbdel 119, 121, **210**
ajbfre 52, 76, 119, 121, **211**
ajbget 52, 76, 119, 121, **212**
ajbgsz 119, 121, **213**
ajbia 119, **214**
ajbip 119, **215**
ajbtag 119, 120, **216**
ajcfjlong **217**
ajcfjset **217**
ajcfstkjmp **219**
ajclk **220**
ajdi **221**
ajei **221**
ajend 35, 61, 153, 154, 155, **222**
ajentr 11, 20, 21, 23, 44, **223**
ajevcre 104, 105, 108, 174, 185, **225**
ajevdel 104, 106, **226**
ajevnt 104, 106, **227**
ajevrd 104, 106, **228**
ajevsig 52, 58, 76, 104, 106, **229**
ajevtag 104, 105, **230**
ajevwat 28, 58, 104, 105, **231**
ajexit 16, 24, 44, 45, 147, **233**
ajfatl 147, **234**
ajflagrd **235**
ajflagrddi **235**
ajflagwr **235**
ajgmsg **236**
ajgofs **237**
ajgseg **238**
ajgsreg **239**
ajhook 152, **240**
ajinb 108, **241**
ajint **242**
ajinw **241**
ajinx **242**
ajispm 14, 53, **243**
ajitrp 60, **244**
ajivtr **245**
ajivtw 53, 68, **246**
ajivtx **247**
ajlcre 139, 142, 144, **248**
ajlhead 139, 144, **249**
ajlinsc 139, **250**
ajlinsh 139, **251**
ajlinsk 139, 142, 144, **252**
ajlinst 139, 144, **253**
ajlmerg 139, **254**
ajlnext 139, 144, **256**
ajlordk 139, 142, **257**
ajlprev 139, **258**
ajlrmvc 139, 142, 144, **259**
ajlrmvh 139, **260**
ajlrmvt 139, **261**
ajltail 139, **262**
ajmau 126, 129, **263**
ajmfre 126, 128, **264**
ajmgeh 126, 130, **265**
ajmget 126, 128, 130, **266**
ajmgsz 126, 129, **267**
ajmhan 126, 130, **268**
ajmodl **269**
ajmset 126, **270**
ajmxcre 112, 113, 116, 174, 187, **271**
ajmxdel 112, 114, **272**
ajmxget 112, 114, **273**
ajmxsnd 33, 36, 40, 52, 58, 76, 112, 113, 116, 118, **274**
ajmxsndp 112, 113, **275**
ajmxtag 112, 113, **276**
ajmxwat 28, 33, 58, 112, 114, 117, **277**
ajoutb **278**
ajoutw **278**
ajproc **279**
ajprocq **279**
ajprvl **281**
ajprvr **282**
ajrbl 135, **283**
ajresum 158, **284**
ajrstl 135, 136, **285**
ajrtl 135, **286**
ajsend 26, 28, 29, 36, 40, 52, 59, 76, 122, **287**
ajsendp 36, **288**
ajsenw 26, 28, 29, 32, 36, 45, **289**
ajsenwp 36, **291**
ajsgnl 32, 52, 57, 76, **293**
ajsgrd **295**
ajsgres **296**
ajsgwat 28, 32, 57, **297**
ajsint **299**
ajsintq **299**

ajsmcre 91, 93, 94, 95, 98, 99, 174, 183, **301**
ajsmdel 91, 93, 99, **302**
ajsmfre 91, 95, **303**
ajsmget 91, 94, **304**
ajsmrls 91, 95, **305**
ajsmrsv 28, 91, 95, 96, **306**
ajsmsig 52, 58, 76, 91, 94, **308**
ajsmtag 91, 93, **309**
ajsmwat 28, 58, 91, 94, 98, 99, **310**
ajsofs **311**
ajsseg **312**
ajssreg **313**
ajsusp 158, **314**
ajtdf 70, 82, 88, **315**
ajtdg 70, 82, **316**
ajtds 70, 82, 85, **317**
ajtick **318**
ajtkcre 25, 173, 178, **319**
ajtkdel 153, **320**
ajtkid **321**
ajtkill 153, **322**
ajtkpry 47, **323**
ajtkstp 153, **324**
ajtksts **325**
ajtktag 25, **326**
ajtktcb 46, **327**
ajtktrm 153, 154, 155, **328**
ajtmcnv 34, 69, 75, 76, 109, 116, 117, **329**
ajtmcre 34, 69, 75, 76, 116, 173, 181, **330**
ajtmdel 34, 69, 75, 76, **331**
ajtmrd 34, 52, 69, 76, **332**
ajtmtag 69, 75, **333**
ajtmwr 34, 52, 69, 75, 76, 116, 181, **334**
ajtrig 26, 28, 29, 52, 59, 76, 116, 118, **335**
ajtslv 70, 81, 179, **336**
ajtsof 70, 79, 81, **337**
ajtson 70, 79, 81, **337**
ajupt **338**
ajver **339**
ajwait 28, 32, 57, 103, 158, **340**
ajwakc 32, **341**
ajwakcs **342**
ajwake 32, 52, 57, 76, 103, **343**
ajwapr **344**
ajwatm 28, 32, 34, 57, **345**

AMBKOF 161
AMBKON 161
AMHALT 149

AMX Library 193, 201, 202, 203, 204
AMX ROM Option/Access
(Refer to AMX Tool Guides)
AMX Service Procedures
Beginning on page 193
Ending on page 345
AMX Tool Guide 2
Arithmetic overflow 60, 61, 62, 63, 147
Array bounds violations 60, 61, 62, 63, 147
Assembler calls to AMX 381–94
Assemblers
(Refer to AMX Tool Guides)
Assembly language
programming in 203, 204

B

Boolean flags
(see Event group, flags)
Bounds checking 60, 61, 62, 63, 147
Breakpoint Manager 27, 147, **159**, **160**, **161**, **162**, 171, 191
compatible debuggers 162
entry delay 191
exclusion 160
exit delay 191
exit detection 160
interrupt masking 161
NMI as breakpoint 162
Buffer 119–24
free 119, 121
get 119, 121
ownership 18, 119, 121, **122**, **123**, **124**
size 120, 121
use count 120–24
Buffer Manager 18, 119–24
Service Procedure Summary 197, 391
Buffer pool 3, 18, 119–24
create 120
definition structure 369, 377
delete 121
find id 120
id 3, 119, 120, 190
maximum number 119, 120, 174
number of buffers 119, 120
predefine using Builder 163, 189, 190
storage allocation 120
tag 120, 190
wait queue 18
Building an AMX system
(Refer to AMX Tool Guides)
(see also Configuration Builder)

C

C language interface
Service Procedure Summary 199
C programming 202, 203, 204
Calendar clock 70, 82–89
(see also Time/Date Manager)
calloc 125
Century 84, 85, 88, 89
Circular list 3, 19, 135, 136, 137, 138
element 4, 135, 136, 137, 138
ISP use 52
slot 5, 135, 136, 137, 138
storage size 136, 137, 138
structure 137, 138
Circular List Manager 19, 135, 136, 137, 138
Service Procedure Summary 198, 394

Class of AMX service
buffer pool, buffer 197
C language interface 199
circular list 198
counting semaphore 196
event group 196
interrupt control 194
linked list 198
memory allocation 197
message exchange 197
PC Supervisor 200
processor support 199
resource semaphore 196
system control 194
task control 195
task termination 195
time/date 196
timing control 196
Clock frequency 71, 172
Clock Handler 3, 15, 34, 65, **71**–**74**
Clock installation 71–74
Clock Interrupt Handler **71**–**74**
Clock ISP 15, 65, **71**–**74**
Clock ISP root **71**–**74**
Clock tick 3, 15, **69**, 172
(see also System tick)
Compilers
(Refer to AMX Tool Guides)
Compute bound 35, 85, 157
Concurrent execution 9
Configuration Builder 163–91
Configuration Generator 53, **164**, **165**, **166**, 353–63
porting 164, 353–63
Configuration Manager 164–91
field editing 168, 169
menu selections 168
Configuration module
(see System Configuration Module)
Conforming ISP
(see also Clock ISP)
(see Interrupt Service Procedure, conforming)
Constant definitions 365–79
Constructing an AMX system
(Refer to AMX Tool Guides)
(see also Configuration Builder)
Context switching 13, 14, 15
Convert ms to system ticks 75
Coprocessor 101, 102, 152, 162
Critical code section 64, 157

D

Day of week 85, 88, 89
Debugger
(see Breakpoint Manager)
Debugging 159, 160, 161, 162
Debugging aid 46, 150, 151
Delay 34
Development tools
(Refer to AMX Tool Guides)
Device input/output services 199
DGROUP (see Groups)
Disable interrupts 199
Divide by zero 60, 61, 62, 63, 147
Download
(Refer to AMX Tool Guides)

E

Enable interrupts 199
Envelope
(see Message envelope)
Error
bounds 50
divide 50
overflow 50
Error codes 3, 150, 151, **349**, **350**
Error Procedure 150, 151, 175
Event group 3, 17, 58, 103–9
create 104, 105
delete 105
find id 105
flags 103, 105
id 3, 104–9, 186
initial state 105
ISP/task synchronization 58
maximum number 105, 174
predefine using Builder 185, 186
read 105
signal 17, 103, 105, 107, 108, 109
state 105
status 105
synchronization using 17, 105–9.
tag 105, 186
wait for event(s) 17, 105, 107, 108, 109
Event Manager 17, 33, 58, 103–9
Service Procedure Summary 196, 389
Exception traps 60, 61, 62, 63, 68
(see also Task, traps)
Exit
(see System, shutdown)
Exit codes
(see Fatal exit codes)
Exit Procedures 3, 16, **44**, **45**, 176, 177
interrupt state 44, 45
stack size 44, 45

F

FAR pointers 204
Fatal error 3
Fatal exit 3, 60, 147, 148, 149, 175
Fatal exit codes 147, 148, 149, **351**
Fatal Exit Procedure **147**, **148**, **149**, 175
  interrupt state 148
  stack size 148
Faults (processor exceptions) 68
Fences (stack) 46
File names (AMX files) 7
Flags
  (see Event group, flags)
free 125
Function prototyping 201

G

Group id
  (see Event group, id)
Groups
  DGROUP 31, 42, 44, 55, 62, 76, 86, 132, 148, 151, 156, 180
Grow a memory block 129

H

Halt 149
Handle 3
  (see also Memory pool, handle)

I

I/O, device input/output services 199
Id variables 163
  buffer pool 190
  event group 186
  message exchange 188
  semaphore 184
  task 179, 180
  timer 182
Installation
  clock 71–74
  ISP 53, 68
Instruction counting 34
Integer size 7
Interrupt
  device 50
  disable by ajdi 199
  enable by ajei 199
  external 14, **49**, **51**, 53, 56, 64, 65, 68
  initial state 20
  masking by debugger 161
  nested 14, 51, 52, 56
  non-maskable (NMI) 50
  response 14, 31, 49, 65
  state 31, 42, 43, 44, 45, 51, 76, 77, 86, 87, 148, 150, 151
Interrupt control
  Service Procedure Summary 194, 386
Interrupt controller 161, 162
Interrupt Descriptor Structure 371, 379
Interrupt Handler 3, 4, 14, **51**–**59**, 65, 66, 67, 68
  clock
    (see Clock Interrupt Handler)
  shared 66, 67
Interrupt Service Procedure 4, 14, **49**–**68**
  conforming 3, 14, **51**–**59**, 65, 66, 67, 68
  installation 53, 68
  nonconforming 5, 14, **51**, 64, 65, 68
  non-maskable (NMI) 64, 68
  stack size 51, 56, 65
  task synchronization 57, 58, 59
Interrupt Stack 14, 51, 56, 65, 171
Interrupt Supervisor 14, 49–68
  Service Procedure Summary 194
Interrupt Vector Table 68
ISP
  (see Interrupt Service Procedure)
ISP root 3, 4, 14, **51**, **52**, **53**, 65, 66, 67, 68
  structure definition 371, 379
ISP root, clock 71–74

K

Kernel Stack 42, 43, 65, 76, 77, 85, 86, 87, 170
Kernel Task 4, 15, 30, 31, 34, 47, 57, 58, 71, 75, 76, 77, 81, 83, 85, 158, 170
  priority 30, 47
  tag 47
Key
  (see Linked list, key)
Key node
  (see Linked list, key node)

L

Launch 20, 21, 22, 23
  parameter 20–24, 20–24, 365, 373
  permanent 20, 21, 22
  temporary 20, 23
Leave
  (see System, shutdown)
Librarian
  (Refer to AMX Tool Guides)
Library, AMX 193, 201, 202, 203, 204
Linked list 4, 19, 139–46
  create 142, 143, 144, 145
  head 140
  header 140, 142, 143, 144, 145, 370, 378
  ISP use 52
  key 139–46
  key node 140, 142, 143, 144, 145, 370, 378
  node 140–46, 370, 378
  node offset 140, 142, 143, 144, 145
  nomenclature 140
  object 139–46
  tail 140
Linked List Manager 19, 139–46
  Service Procedure Summary 198, 393, 394
Linkers and Locators
  (Refer to AMX Tool Guides)

M

Mailbox 4, 26
(see Message (task), mailbox)
acknowledge message 35
message queue 4
Make utilities
(Refer to AMX Tool Guides)
malloc 125
Math coprocessor 101, 102, 152
Memory addressing (24-bit) 1, 125, 171
Memory allocation 18, 125, 126, **128**–**34**
Memory assignment
(see Memory Assignment Procedure)
Memory Assignment Procedure **131**–**34**, 175
Memory block 4, **125**–**34**
find size 129
free 18, 125, 126, 128, 130
get 18, 125, 126, 128, 130
header 127
ownership 18, 125, 126, 127, **128**, **129**
resize (grow or shrink) 129
use count 127, 128, 129
Memory handle 4
(see also Memory pool, handle)
Memory management unit 152
Memory Manager 18, **125**–**34**
24-bit memory 125
assign memory to 126, 128, 131–34
nomenclature 127
Service Procedure Summary 197, 392
Memory paging 152
Memory pool 18, **125**–**34**
assign memory to 131
handle 127, 130
maximum number 125, 128
memory assignment 132, 133, 134
private 125, 130
section 18, **125**–**34**
size 125, 128
Memory section 4
(see Memory pool, section)
Message 4
alignment 36, 40
from ISP 58
passing 16, **36**–**41**, 58, 36–41
priority 5, 16
(see Message (task), priority)
(see Message exchange, message priority)
size 36, 37, 40, 113
Message (task) 25, 26, 29, 31, **36**–**41**
acknowledge message 35
extension 41, 367, 375
id 180
mailbox 18, 29, **36**–**39**, 180
predefine using Builder 180
priority 36–39
queue depths 180
receive **36**–**41**
send 36–38, **36**–**41**, 59
tag 180
Message envelope 3, 16, **36**–**41**, 111, 113, 114
maximum number 170
size 37, 170
Message exchange 4, 16, 18, 36, 37, 38, 39, 58, **111**–**18**
create 113
delete 114
find id 113
id 4, 112, 113, 188
ISP/task synchronization 58
maximum number 113, 174
message priority 5, 16, 18, 36, 37, 38, 39, 111–14
message queue 16, 18, 36, 37, 38, 39, **111**–**14**, 188
predefine using Builder 187, 188
queue depths 113, 188
send message to 58, 111–18
synchronization using 18, 111–18.
tag 113, 188
wait for message 58, 111–18
wait queue 18, 111, 112, 113, 114
Message Exchange Manager 18, 33, 58, 111–18
Service Procedure Summary 197, 390
Message priority
(see Message (task), priority) .
(see Message exchange, message priority)
Message queue 5
(see Message (task), mailbox)
(see Message exchange, message queue)
Message send
(see Message (task), send/receive)
(see Message exchange, send to/wait for)
Model
Large 202
Medium 171, 180, 202
Multitasking 9

N

Names, AMX reserved 7, 347
NMI (non maskable interrupt) 162
NMI (Non-maskable interrupt) 64, 68
Node
(see Linked list, node)
Nomenclature 7, 201, 202, 203, 204
Nonconforming ISP
(see Interrupt Service Procedure, nonconforming)
Non-maskable interrupt (NMI) 64, 68
Numeric coprocessor 101, 102, 152, 162

O

Overflow (arithmetic) 60, 61, 62, 63, 147
Ownership
buffer 119, 121, 122, 123, 124
memory block 125, 126, 127, 128, 129
resource 95, 101, 102

P

PC Supervisor
(see PC Supervisor Reference Manual)
Service Procedure Summary 200
Preemption 13
Priority
(see Event group, wait for event(s))
(see Kernel Task, priority)
(see Message exchange, message priority)
(see Message exchange, wait for message)
(see Semaphore, counting, wait)
(see Semaphore, resource, reserve)
(see Task termination, deletion priority)
(see Task, priority)
Priority scheduling 30
Privileged
(see Task, privileged)
Procedures, AMX
Beginning on page 193
Ending on page 345
Procedures, AMX (summary of) 193
Processor exceptions 50
Processor exceptions (traps) 60, 61, 62, 63, 68
(see also Task, traps)
Processor initialization 20, 42, 43, 44, 45
Processor support
Service Procedure Summary 199
Programming in assembler 381–94

Q

Queue
(see also Circular list)
(see also Linked list)
(see Event group, wait queue)
(see Message (task), mailbox)
(see Message exchange, message queue)
(see Message exchange, wait queue)
(see Semaphore, wait queue)

R

RAM (random access memory) 5, 7, 21, 68
Real-time clock
(see Clock)
Reentrant code 66, 67, 123, 125, 135, 139, 194
Register array 371, 379
Reserved words 7, 347
Resolution
(see Timer, resolution)
Resource management 17
Resource ownership 95, 101, 102
Resource semaphore
see Semaphore, resource
Restart Procedures 5, 11, 20, 42, 43, 176, 177
interrupt state 42, 43
stack size 42, 43
Resume task 158
ROM (read only memory) 5, 7, 21, 68
ROM Option/Access
(Refer to AMX Tool Guides)
Round robin scheduling 70

S

Scheduler hooks 152
(see also Time/Date Scheduling Procedure)
Scheduling algorithm 13, 29, 30, 70, 79, 80, 81
Section
(see Memory pool, section)
Segment 5
Segment register array 371, 379
Segment registers 371, 379
Semaphore 5, 17, 58, 91–102
binary 17, 91, 95
counting 3, 17, 58, 93, 91–94, 97, 98, 99, 100, 184
signal 94
wait 94
create 93, 94, 95
delete 93
find id 93
id 5, 92, 93, 95, 184
ISP/task synchronization 58
maximum number 93, 174
mutual exclusion 94, 97, 98
P and V operations 91
predefine using Builder 183, 184
resource 5, 17, 91, 92, 93, **95–96**, 101, 102, 184
nesting 95, 101, 102
release 95
reserve 95, 96
synchronization using 99, 100.
tag 93, 184
wait queue 17, 95, 96
Semaphore Manager 17, 33, 58, 91–102
Service Procedure Summary 196, 388
Send message
(see Message (task), send/receive)
(see Message exchange, send to/wait for)
Service Procedure Summary 193–200
Service Procedures, AMX
Beginning on page 193
Ending on page 345
Shrink a memory block 129
Software development
(Refer to AMX Tool Guides)
Stack checking 46
Stack fences 46
Stack overflow 46

Stack size
Exit Procedures 44, 45
Fatal Exit Procedure 148
Interrupt Service Procedure 51, 56, 65
Interrupt Stack 51, 56, 65, 171
Kernel Stack 42, 43, 65, 76, 77, 85, 86, 87, 170
Restart Procedures 42, 43
task 44, 45, 51, 61, 65, 155, 156, 179
Task Termination Procedure 155, 156
Task Trap Handler 61
Time/Date Scheduling Procedure 85, 86, 87
Timer Procedure 76, 77
User Error Procedure 150, 151
User Scheduler Hooks 152
Structure definitions 365–79
Suspend task 158
Symbols, AMX reserved 7, 347
Synchronization 32, 33
(see Event group, synchronization)
(see Message (task), synchronization)
(see Message exchange, synchronization)
(see Semaphore, synchronization)
(see Task, signals)
(see Task, wait/wake synchronization)
System
parameters 170–75
ROMed 68, 171
shutdown 16, **23**, 44, 45, 147
startup 11, **20**, **21**, **22**, **23**, 29, 42, 43
System Configuration Module 5, 11, 21, 22, 23, 163–68, 353, 163–68
System Configuration Template 164, 165, 166, 353, 359–62
System control
Service Procedure Summary 194, 383
System Documentation Module 164, 165, 166
System Documentation Template 164, 165, 166
System generation process 163–91
System tick 5, 15, 69, 172
(see also Clock tick)
convert ms to 75

T

Tag 5
Task 6, 11, 25–35
breakpoint 159
create 25
definition structure **365**, 373
delay 34
delete 154, 155, 156, 157
deletion priority 157
end 13, **35**, 81, 153, 154, 155, 156
execution 13, **29**, **30**, **31**, 59
find id 25
halt 159
id 6, **25**, 154, 179
interrupt state 31
ISP synchronization 57, 58, 59
kill 154, 155, 156, 157
mailbox
(see Message (task), mailbox)
maximum number 25, 173
periodic execution 29
predefine using Builder 163, 178, 179, 180
priority 6, **30**, 35, 47, 79, 80, 81, 179
privileged 31
resume 158
signals 6, **57**
stack 26, 31, 35, 44, 45, 46, 51, 61, 65, 155, 156, 163, 179
stack size 44, 45, 51, 61, 65, 179
start 26, 29, 31, 35, 81
state 11, **27**, **28**, **27**–**35**, 69, 27–35
halt 27
idle 27–29
ready 27–29
run 27, 28
wait 27, 28, 37
status block 366, 374
stop 153, 154, 155, 156, 157
suspend 158
synchronization 32, 33
(see also Synchronization)
tag 25, 46, 179
time slice 5, 70, 79, 80, 81, 179
timed wait 34
timeout 32, 34
timing 15, 34, 69, 70, 71, 79, 80, 81
traps 60, 61, 62, 63, 147
trigger 26, **29**, 35, 59, 154
wait/wake synchronization 57, 103
Task control
Service Procedure Summary 195, 384, 385
Task Control Block (TCB) 6, 25, 46, 152, 365, 373
reserved for user 46, 152
Task Mailbox
(see Message (task), mailbox)
Task Scheduler 13, 14, 29, 30, 35, 52, 56, 70, 79, 80, 81, 152, 153
user hooks 152
Task switching 51, 52, 56

Task termination 153, 154, 155, 156, 157
delete 154, 155, 156, 157
deletion priority 157
enable/disable 154
kill 154, 155, 156, 157
Service Procedure Summary 195
stop 153, 154, 155, 156, 157
Task Termination Procedure 154, 155, 156, 366, 374
Task Trap Handler 60, 61, 62, 63, 147
stack size 61
TCB (see Task Control Block) 6
Testing 159, 160, 161, 162
Tick
(see Clock tick)
(see System tick)
Time slice 6, 70, 79, 80, 81
enable/disable 79, 80, 81, 172
interval 5, 79, 80, 81
Time/Date
ASCII formatting 88, 89
century 84, 85, 88, 89
day of week 85, 88, 89
default 83
get 82, 83
set 82, 83, 85
structure 84, 86, 87, 369, 377
timer 83
validity 85, 88, 89
Time/Date Manager 17, 70, 82–89, 172
Service Procedure Summary 196, 387
Time/Date Scheduling Procedure 82, 83, 85, 86, 87, **82**–**87**, 172
interrupt state 86, 87
stack size 85, 86, 87
Timer 6, 11, 34, 69–78
(see also Time/Date, timer)
create 75
delete 75
find id 75
id 6, 75, 182
interval 75, 76
maximum number 75, 173
one-shot 75
parameter 69, **75**, **76**, 182
periodic 15, 29, 69, 71, 75, 76
predefine using Builder 181, 182
resolution 34
start 75
stop 75
tag 75, 182
Timer Manager 15, 34, 69–89
Service Procedure Summary 196, 387
Timer Procedure 6, 15, 34, 69, **75**, **76**, **77**, **78**, 182
interrupt state 76, 77
parameter 69, **75**, **76**, 182
stack size 76, 77
Timing control
(see Timer Manager)
Tools
(Refer to AMX Tool Guides)
Traps
(see Task, traps)
Turbo86 (VAutomation) 1, 125, 171

U

Underscore in names 100, 172
User Error Procedure 150, 151, 175
User Parameter File 164–69, 353–63
User Parameter Table 21, 22, 23, 21–24, 21–24, 163, 368, 376
User Scheduler Hooks 152

V

Variables, id 163
(see also Id variables)
VAutomation 1, 125, 171
Vector Table 68
Volatile variable access 199

W

Wait queue
(see Event group, wait queue)
(see Message exchange, wait queue)
(see Semaphore, wait queue)
Warning codes 150, 151, 350